Table of contents

1. Introduction

Most of this material is derived from the text, Han, Kamber and Pei, Chapters 4 and 5, or the corresponding powerpoint slides made available by the publisher. Where a source other than the text or its slides was used for the material, attribution is given. Unless othewise stated, images are copyright of the publisher, Elsevier.

Here, we continue from the previous e-book on Data Warehousing to work on OLAP and cubes.

ACTION: Watch this video for a brief overview of the topic if you wish.

Analysing Data in Warehouses: Overview

2. Data Mining in Data Warehouses (Text:4.3)

Three kinds of data warehouse applications *(in order of increasing sophistication)*

(1) Information processing

- supports querying, basic statistical analysis, and reporting using crosstabs, tables, charts and graphs
- provides information directly stored or aggregated.

(2) Analytical processing: *On-line analytical processing (OLAP)*

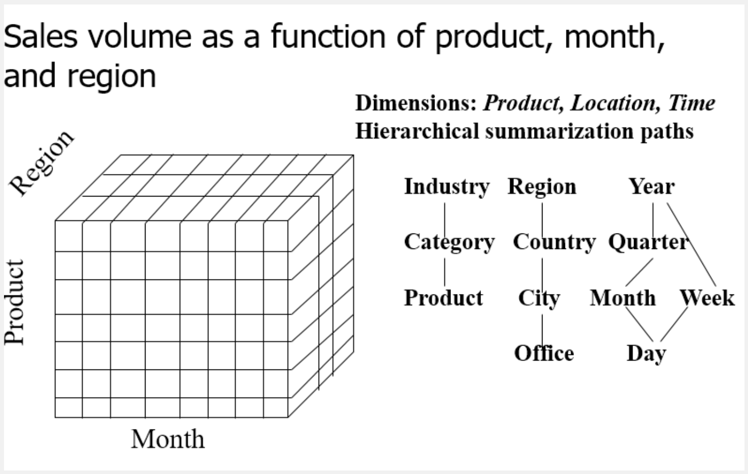
- *multidimensional* analysis of data warehouse data.
- supports basic OLAP operations, slice-dice, drilling, pivoting.
- derives information summarised at multiple granularites from user specified subsets (i.e user-specified concepts).

(3) multidimensional data mining: also called *Exploratory multidimensional data mining*, *Online Analytical Mining*, and *OLAM*

- knowledge discovery from hidden patterns
- supports finding associations, constructing analytical models, performing classification and prediction, and presenting the mining results using visualization tools
- *automated* discovery of implicit patterns
- integrates OLAP with data mining; OLAP can be used for inital exploration
- benefits from the high quality of data warehouse data
- information processing infrastructure of the data warehouse can be used (access, APIs, storage, security, maintenance etc).

3. Typical OLAP Operations (Text 4.2.5 + 4.4.4)

In the multidimensional model, data are organised into multiple dimensions and each dimension contains multiple levels of abstraction defined by concept hierarchies.


Sales volume as a function of product, month, and region
Dimensions: Product, Location, Time
Hierarchical summarization paths
Region
Product
Month
Industry
Category
Product
Region
Country
City
Office
Year
Quarter
Month
Week
Day

The typical OLAP operations are implemented in OLAP servers.

The SQL standard also defines some OLAP operators but these are generally implemented inconsistently in relational databases.

A common architecture for multidimensional cube operations is to extend and optimise relational architecture to form a **ROLAP** (i.e Relational OLAP) server which is likely to rely on a *star schema* (+snowflake, fact constellation) database structure. A form of this highly specialised for ROLAP may drop support for common OLTP functions.

Alternatively, a **MOLAP** (Multidimensional OLAP) server uses a very different column-oriented data storage architecture, which is particularly well suited to optimise rapid access to aggregate data and to storage of sparse cubes.

Finally, a hybrid architecture **HOLAP** (Hybrid OLAP) server combines ROLAP and MOLAP, with detailed data in a relational database and aggregations in a MOLAP store. This may have the performance advantages of each.

3.1. Roll-up

Rollup (also called *drill-up*) summarises data in one of two ways.

1. By dimension reduction

Move from one cuboid to another higher up the lattice towards the apex, where some dimension of the cube is removed and the remaining dimensions are represented by measures that are now aggregated over the missing dimension.

Example:

Consider the datacube *item x time*, with the measure being *Sales*

Then starting from the 2-D cuboid here:

Sales	**Month**											
Item	Jan	Feb	Mar	Apr	May	June	July	Aug	Sep	Oct	Nov	Dec
Dell laptop	2	2	5	5	4	6	3	3	7	9	11	4
Dell desktop	5	4	5	4	6	7	2	5	6	6	8	7
Mac Plus	10	6	6	7	5	8	7	5	7	11	7	6
Apple IIe	1	3	1	2	1	3	2	2	3	1	0	1
Commodore 64	0	0	1	0	2	5	0	1	1	0	2	3
Apricot	0	0	2	1	0	4	1	1	2	0	2	2
Sun solaris	5	2	3	4	5	6	4	2	1	3	4	4

Rollup on *item* to get the 1-D cuboid:

Sum of Sales	**Month**											
	Jan	Feb	Mar	Apr	May	June	July	Aug	Sep	Oct	Nov	Dec
All items	23	17	23	23	23	39	19	19	27	30	34	27

2 . By climbing up the concept hierarchy

Move from one cuboid to another by stepping up a level in a concept hierarchy on one dimension. This does not remove a dimension from the cube but aggregates the measures for that dimension into bigger chunks, and so reduces the number of data points along that dimension.

Example:

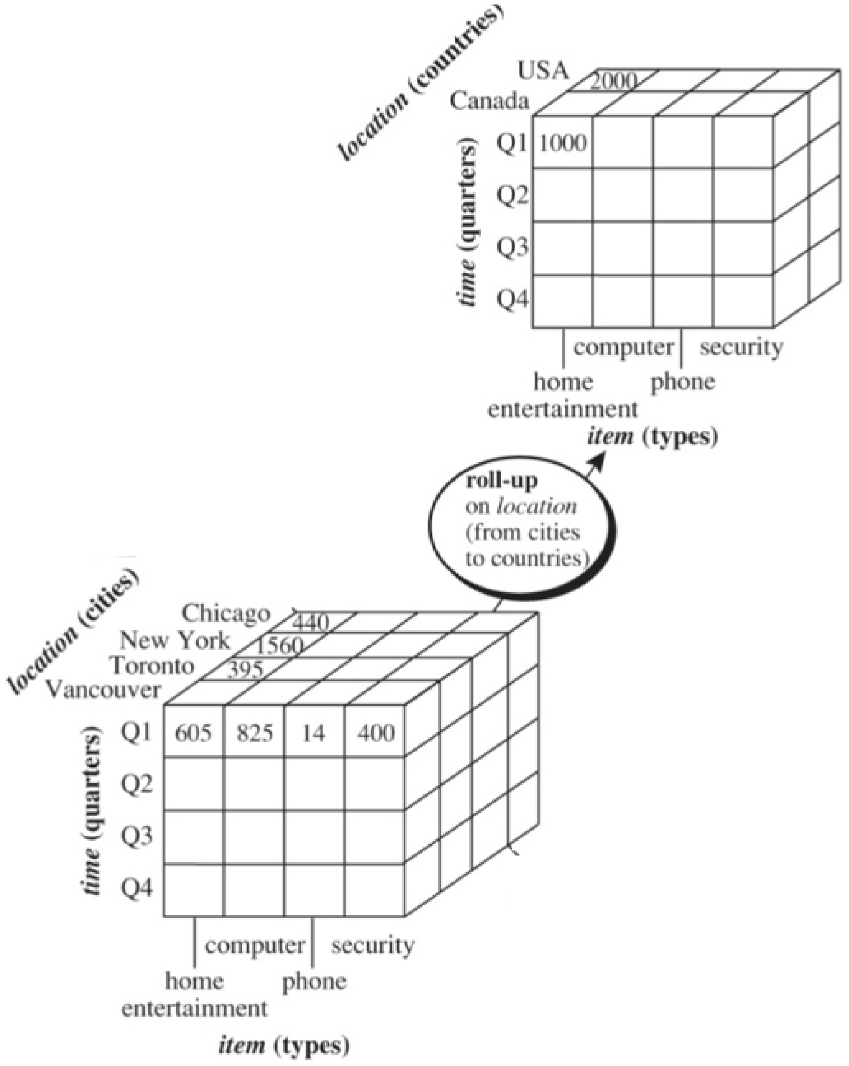
Consider the cube of *sales* with dimensions *item x time x location* and the concept hierarchy along the location dimension of (*offices < cities < countries < regions).*

Starting from the cuboid *types x quarters x cities,* a rollup operation:

***roll-up** on location from cities to countries*

would move to the cubiod t*ypes x quarters x countries*. Now, instead of being grouped by cities, the data is grouped by countries.

Another subsequent rollup along the same location dimension would move to the cuboid *types x quarters x regions.*



location (countries)
USA
Canada
2000
Q1
1000
Q2
Q3
Q4
time (quarters)
home entertainment
computer
phone
security
item (types)
roll-up on location (from cities to countries)
location (cities)
Chicago
New York
Toronto
Vancouver
440
1560
395
Q1
605
825
14
400
Q2
Q3
Q4
time (quarters)
home entertainment
computer
phone
security
item (types)

3.2. Drill-down

Drill-down (also called **roll down**) is the reverse of rollup, navigating from less detailed data to more detailed data. As for rollup, there are two ways to do this.

1. By introducing additional dimensions

Starting from the 1-D cuboid here:

Sum of Sales	Month											
	Jan	Feb	Mar	Apr	May	June	July	Aug	Sep	Oct	Nov	Dec
All items	23	17	23	23	23	39	19	19	27	30	34	27

Drill-down on *item* to get the 2-D cuboid as follows.

Sales	Month											
Item	Jan	Feb	Mar	Apr	May	June	July	Aug	Sep	Oct	Nov	Dec
Dell laptop	2	2	5	5	4	6	3	3	7	9	11	4
Dell desktop	5	4	5	4	6	7	2	5	6	6	8	7
Mac Plus	10	6	6	7	5	8	7	5	7	11	7	6
Apple IIe	1	3	1	2	1	3	2	2	3	1	0	1
Commodore 64	0	0	1	0	2	5	0	1	1	0	2	3
Apricot	0	0	2	1	0	4	1	1	2	0	2	2
Sun solaris	5	2	3	4	5	6	4	2	1	3	4	4

2. By stepping down a concept hierarchy

Move from one cuboid to another by stepping down a level in a concept hierarchy on one dimension. This does not remove a dimension from the cube but disaggregates the measures for that dimension into smaller chunks, and so increases the number of data points along that dimension.

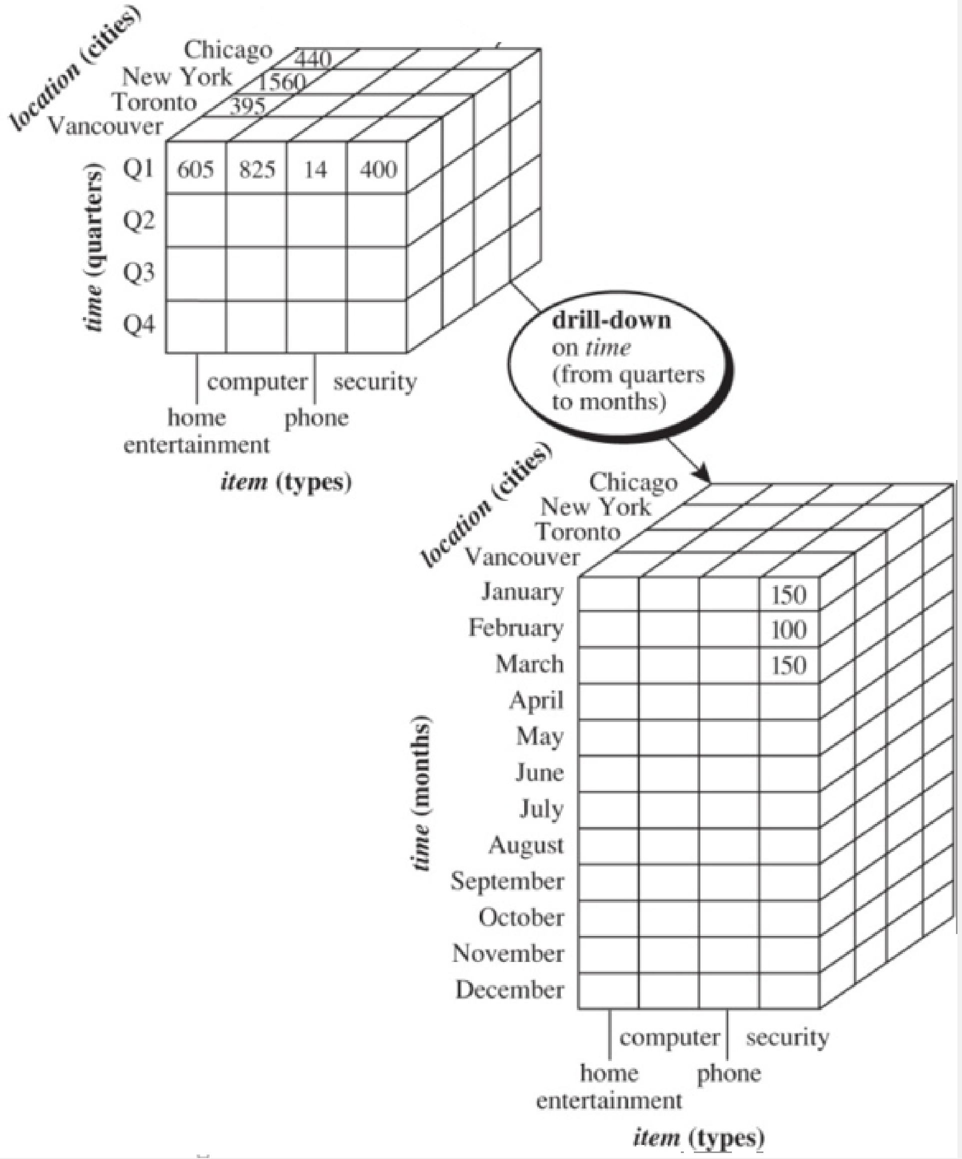
Example:

Consider the cube of *sales* with dimensions *item x time x location* and the concept hierarchy along the *time* dimension of (*days < months < quarters < years)*. Starting from the cuboid *types x quarters x cities,* a drill-down operation:

***drill-down** on time from quarters to months*

would move to the cubiod t*ypes x months x cities.*Now, instead of being grouped by quarters, the data is grouped by months.

Another subsequent drill-down long the same *time* dimension would move to the cuboid *types x days x cities.*



location (cities)
Chicago
New York
Toronto
Vancouver
440
1560
395
time (quarters)
Q1
Q2
Q3
Q4
605
825
14
400
computer
security
home entertainment
phone
item (types)
drill-down on time (from quarters to months)
location (cities)
Chicago
New York
Toronto
Vancouver
time (months)
January
February
March
April
May
June
July
August
September
October
November
December
150
100
150
computer
security
home entertainment
phone
item (types)

3.3. Slice and Dice

Slice and **Dice** cut out rectangular sections of a cuboid.

1. Slice

Slice cuts off one dimension of the cube, not by aggregating but by selecting only one fixed value along one dimension.

It corresponds to a relational algebra *select* (to choose which fixed value from which dimension) then *project* (on the remaining dimensions).

Example

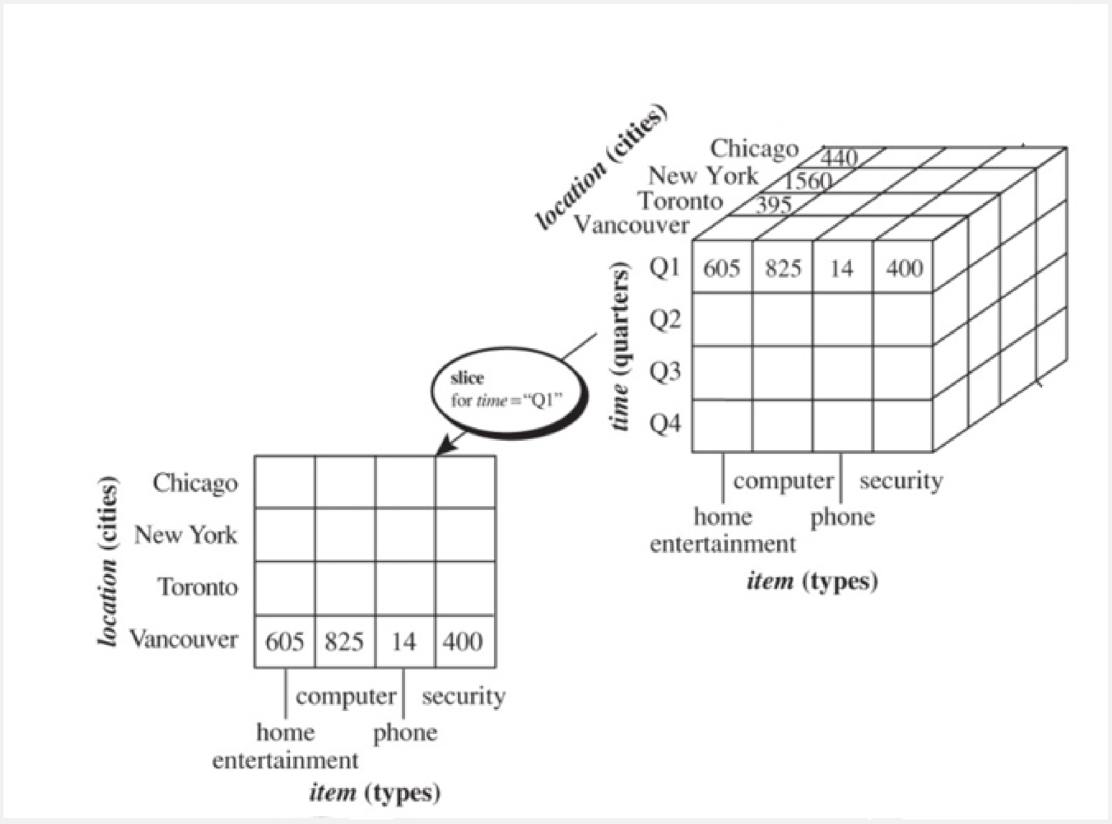
Consider the cube of *sales* with dimensions *item x time x location.*

Starting from the cuboid *types x quarters x cities,* a slice operation:

***slice** for time = "Q1"*

would move to the cubiod t*ypes x cities* and every value represented holds only for the quarter Q1.

Another subsequent slice, say for *item* = "*computer*" would result in a cuboid for *cities* alone, with all data for each city being values only for computers in Q1.



2. Dice

Dice cuts out a sub-cube, not by aggregating but by selecting multiple fixed values for each of multiple dimensions.

It corresponds to some relational algebra *selects* to choose which fixed values from which dimensions.

Example

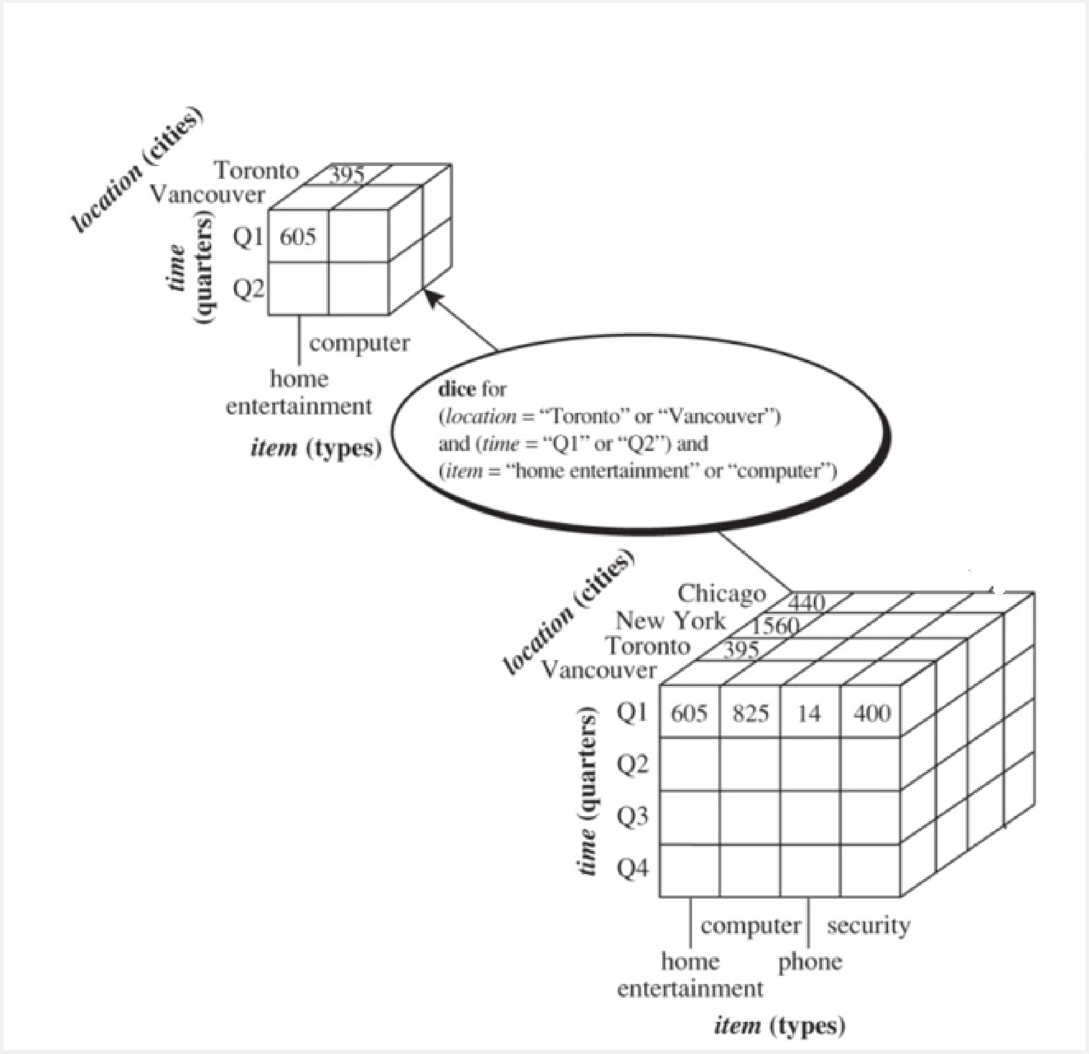
Consider the cube of *sales* with dimensions *item x time x location.*

Starting from the cuboid *types x quarters x cities,* a dice operation:

> ***dice** for (location = "Toronto" or "Vancouver") and (time = "Q1" or "Q2") and (item = "home entertainment" or "computer")*

would move to the cubiod t*ypes x quarters x cities* but only data for those selected dimension values would be represented in the cuboid. That is, the only data in the cuboid is for computers and home entertainment products, in Q1 and Q2, from Toronto and Vancouver.

Another subsequent dice, say for *item* = "*computer*" would result in a 3-D cuboid without any home entertainment products.

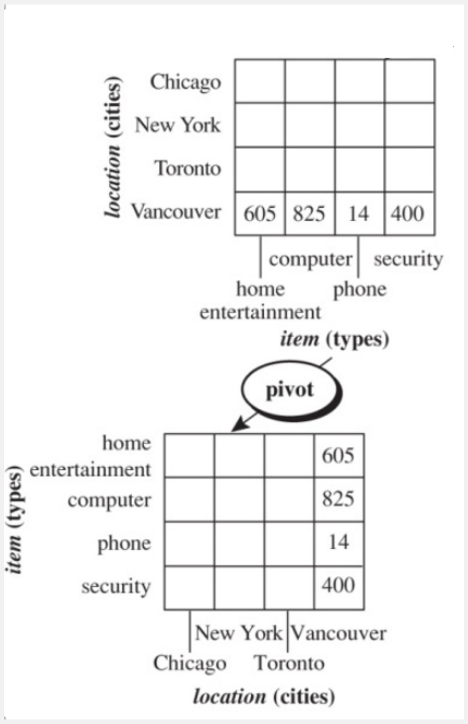


3.4. Pivot and Other Operations

Pivot (also called **rotate**) is a visualisation operator and does not change the data. It changes the data axes in view to an alternative presentation, such as by geometrically rotating a 3D cube or by presenting a 3D cube as a series of 2D planes.

Example

Starting from the 2-D presentation below of sales with dimensions *types x cities,* a *pivot* operation would rotate the axes to show *cities x types* and the values would remain unchanged.



Other OLAP Operations

OLAP servers typically offer many other operations. Some of the more common are

- Drill-across: involving (across) more than one fact table
- Drill-through: through the bottom level of the cube to its back-end relational tables (using SQL)
- Ranking: top-k or bottom-k items in a list ordered by some measure
- Moving averages: over time
- Statistical functions
- Domain specific operators: growth rates, interest, internal rate of return, depreciation, currency conversion.

4. Processing OLAP queries

Data Warehouses contain huge volumes of data, yet aim to answer queries in interactive query time time-frames.

One way to deliver fast response times is to pre-compute all the aggregate measures required, at a much increased storage cost.

Data Warehouses must support efficient cube computation, access methods and query processing techniques.

4.1. Efficient Data Cube Computation (Text:4.4.1)

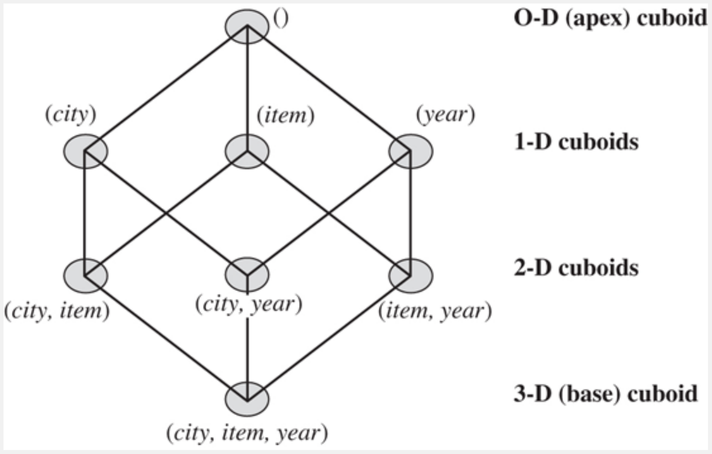
Data Cubes

- A Data cube can be viewed as a lattice of cuboids where each cuboid represents a choice of group-by attributes.
- The bottom-most cuboid is the base cuboid
- The top-most cuboid (apex) contains only one cell

Example:

The diagram shows a lattice of cuboids making up a 3D data cube of *sales* with dimensions *city x item x year*. Each cuboid represents a different group-by.

The base cuboid contains all three dimensions, at the lowest level of aggregation, that is, the finest granularity.



- The 3-D base cuboid represents the answer to queries such as "*What are the sales for each city, item and year?*". It is the least generalised (most specific) cuboid.
- The 2-D cuboids represent the answer to queries such as "*What is the sum of sales, grouping by city and item?*".
- The level-2 1-D cuboids represent the answer to queries such as "*What is the sum of sales, grouping by city?*".
- The 0-D apex cuboid, often denoted all, represents the the answer to "What is the sum of sales?". It is the most generalised (least specific) cuboid.

- The total number of cuboids for this 3-D data cube is $2^3 = 8$.

Compute cube

This cube may be only conceptual, but it can also be *materialised* (that is, pre-computed) in full to reduce run-time query processing costs.

It can be generated by a sequence of SQL group-by queries, one for each cuboid.

In a generalised syntax, the complete data cube above could be defined by the query

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

And it could them be fully materialised as all 8 cuboids by

compute cube sales_cube

In extended SQL by

```
SELECT city, item, year, SUM(sales in dollars)
```

```
        FROM SALES

        GROUP BY CUBE  (city, item, year)
```

However, the full materialisation can be useful for fast query response times, but very expensive in storage.

Size of the materialised data cube

In general, the number of cuboids in a data cube of dimension n is $\prod_{i=1}^{n}(1+1) = 2^n$. This is simply all combinations of the binary decision for each dimension to aggregate, or not to aggregate. [N.B. π here means the product of the expressions represented by varying the index value, i, from 1 to n]

However, when we also count cuboids generated for concept hierarchies for roll-ups, for L_i being the number of levels in the hierarchy for concept i, we get the total number of cuboids, T,

$$T = \prod_{i=1}^{n} (L_i + 1)$$

Each dimension in a cuboid can have only one concept level represented so we need a fresh cuboid for each L_i, plus 1 more to account for aggregation right across the whole dimension (the summary for that dimension), irrespective of the number of levels for the dimension (because the aggregation is the same regardless of the concept level).

Action: Pause and check this formula works for the case that there are no concept hierarchies.

Example

Consider a cube of 10 dimensions with 4 levels for each dimension (aggregation over the dimension is not counted as a level). Then the number of cuboids is $5^{10} \approx 9.8 \times 10^6$

ACTION: try this exercise

Exercise: How big is a data cube?
Solution to Exercise: How big is a data cube?

In practice, 60 or so dimensions is not unusual.

As the number of dimensions, the number and depth of conceptual hierarchies, and the number of distinct values in each dimension grows, the storage space for the cuboids can grossly exceed the storage size for the initial data. Furthermore the computation cost of evaluating all cuboids becomes prohibitive.

4.2. Processing OLAP queries (Text: 4.4.3)

Precomputing all of a data cube is often prohibitively expensive in storage, and typically large parts of the cubiod are never used. We can

- Materialise every (cuboid) (full materialisation),
- none (no materialisation),
- or some (partial materialisation)

The latter is the typical choice, but it requires intelligent strategies for run-time query processing.

In general, query processing must

1. **Determine which operations should be performed on the available cuboids**

Transform drill, roll, etc. using corresponding SQL and/or OLAP operations, e.g., dice = selection + projection on a cuboid

2. **Determine which materialised cuboid(s) should be selected for the OLAP operation**

Example
Consider *sales_cube [time, item, location]* with measure *sum(sales_in_dollars)* with 3 hierarchies
time: (day < month < quarter < year)
item: (item_name < brand < type)
location: (street < city < province_or_state < country)

Let the query to be processed be on {brand, province_or_state} with the condition "year = 2017", and there are 5 materialised cuboids available:
1) {year, item_name, city}

2) {year, brand, country}

3) {year, brand, province_or_state}

4) {item_name, province_or_state} where year = 2017

5) {item_name, province_or_state}

Which cuboid could be selected to process the query?

- **The cuboid must contain at least the dimensions mentioned in the query (or more)**
  - *Cuboids 1,2,3 meet this condition*
  - **Although if the selection clause in a query is weaker than the selection clause in the cuboid, and the other dimensions in the query are contained in the cuboid, then it can be used**
    - *This allows cuboid 4 to be used, too, because the dimensions {item_name, province_or_state} mentioned in the query are in the cuboid.*
- **Finer-granularity data cannot be generated from coarser-granularity data (ie higher up the concept hierarchy than the query).**
  - Cuboid 2 cannot be used because *country* is more general than *province_or_state*. That leaves 1,3,4 as the only cuboids that *can* be used.

Which cuboid would be best to use for the query?

- **Prefer a cuboid at the coarsest granularity of data**
  - Cuboid 1 is the finest granularity and should typically not be used. So that leaves 3 and 4.
- **Prefer a small cuboid or one with prebuilt efficient indexes**

    - If there are few *year* values overall but there are many *item_name*s for each *brand* then 3 would be smaller and so a good choice.
    - On the other hand, if 4 has efficient indexes, it could be the better choice.

4.3. Cube Materialisation (Text: 5.1)

The major strategies precomputing cubes are

(1) **Full Cube materialisation** - possibly using the Multiway method, that computes aggregations simultanously across multiple dimensions.

(2) **Cube shell**:

- Compute all cuboids with *k* dimensions or less (for some small *k*, like 3) on the assumption that most investigations only need a few selected dimensions; or
- Compute *shell fragments* by pre-computation and and fill them in with run-time computation: a semi-online strategy

(3) **Iceberg Cube:** The Iceberg is a partial cube that can be built top-down by the *BUC metho*d or top-down and bottom-up by *Star Cubing*.

4.4. Iceberg Cubes (Text: 5.1.1)

Sparse data cubes

Many cells in a cuboid may have a zero measures, and be of little interest to anyone. For example, may stores may not stock many products of the product line, and their counts of items sold for unstocked products will always be zero.

Cuboids with many zero-valued cells are **sparse cuboids**; a cube with many sparse cuboids is a **sparse cube.** Typically, datacube representations are designed for sparse cubes and cells with zero measures are not materialised.

Iceberg Cubes

Further, *low* values of measures are often considered of little interest. If measures are lower than some predefined threshold then such cells may also be left out of the datacube, and although an analyst may not know exactly what the values are, they can infer that they are below that threshold.

Such partially-materialised cubes are called **iceberg cubes** because only the interesting tip of the huge iceberg of data is visible in the cube.

The threshold is called the **minimum support threshold** or *min_sup*.

Example: precomputing an iceberg cube from the *salesinfo* relation

> *compute cube sales iceberg as*
>
> *select month, city, customer group, count(*)*
>
> *from salesInfo*
>
> *group by cube(month, city, customer group)*
>
> *having* ***count(*) >= min_sup***

The constraint "*count(*) >= min_sup*" is called the **iceberg condition**, in this case applying to the count measure, but other measures can also be used.

This approach does avoid materialising trivial aggregate cells, but there are still many redundant cells generated.

Example

Consider the iceberg cube with a 100-D base cuboid of only two cells meeting the threshold condition, say *sum >= 10*. These 2 cells are

A: $(a_1, .., a_{100})$ with sum 10; and

B: $(b_1, ..b_{100})$ with sum 10 ;

and all the $a_i b_i$ for $i = 1, ..n$. That is, for every dimension, the attribute values for A and B are different.

Then, each of the 100 99-D cuboids up one level from the base also have 2 cells in them, because whichever dimension is aggregated at that level, all of the 99 values for the remaining 99 dimensions are different for the two cells with measure value 10. For every level up the cuboid this same pattern will apply; the two cells will continue to be represented in two different aggregate cellls in every cuboid, except only for the apex cuboid. Even at one level below the apex the 100 1-D cuboids will each have two cells because they each have different values for the 1 dimension of every cuboid. Only at the apex, the cells will combine to form a single aggregate with measure value 20.

That is, even though each cuboid is small, with only 2 cells per cuboid (apart from the apex), there are neverthless 2^{100} cuboids to be computed and stored. That is, $2 \times 2^{100} - 1$ cells are stored: 2 for each of 2^n cuboids less 1 for the apex cuboid that has only one cell.

Closed cubes

Clearly, there is a lot of redundant information here, and this too can be reduced through the concept of *closed cells*. An iceberg cube would typically materialise only closed cells. Any datacube, whether using threshold pruning or not, can materialise only closed cells to save storage space.

Non-closed cells can be pruned away from the materialised cube for storage and then re-constructed during query processing. The idea is that any cell only needs to be represented in its least aggregated (lowest) cuboid in the lattice where it satisfies the threshold condition, and there is no need to store it in aggregated form with the very same measure value at any higher level, until it gets aggregated with some other cells that *do* cause the measure value to change. In our example, we would store the two very different cells in the base cuboid, both with sum value 10, and would store the apex cell where those two cells are combined to give sum 20, but no other cells are materialised. The closed iceberg cube then is storing only 3 cells, contrasting with $2^{101} - 1$ for the basic iceberg cube as described above.

A cell is a **closed cell** if there is no other cell that is a descendant in the lattice that has the same measure value as itself. A *descendant* is below it in the lattice; less aggregated; higher dimensionality; with a measure value that contributes to the measure of the closed cell because it has, for all the non-aggregated dimensions of X, the same attribute values as X.

A **closed cube** materialises only closed cells.

ACTION: Watch this video for a worked-through-example

Explaining iceberg cubes

For **query processing**, if we are looking for the measure of some cell that is not materialised, we can look downwards for its highest descendant in the cuboid lattice and use the measure value found there.

Example:

In our example above of two cells A and B at the base, assume we are querying for, say $(*, a_2, *, a_4, .., a_{100})$, where $*$s here indicates aggregation over the first and third dimensions, and the a_is are as before. That is, it is a particular cell in a 98-D cuboid of the datacube. Then, when we find that cell is not materialised in the appropriate 98-D cuboid, we can traverse downwards to check the two 99-D cuboids where the first or third dimension is unaggregated. Here, we look for any materialised descendent cells that match $(?, a_2, *, a_4, .., a_{100})$ in one cuboid and $(*, a_2, ?, a_4, .., a_{100})$ in the other (where '?' is used here to mean to check for every attribute value for that dimension). If we find one, then it has the same value as $(*, a_2, *, a_4, .., a_{100})$ that we are looking for, because it is a descendent and materialised, so we choose that sum value for our answer. But in our example we won't find any answers there as those cells are not closed and therefore not materialised either. Traversing down again to the 100-D cuboid, we look for cells of the form $(?, a_2, ?, a_4, .., a_{100})$ and we find the non-closed cell $(a_1, a_2, a_3, a_4, .., a_{100})$ which is A and so we have our answer as the sum value of A, which is 10.

5. Cube Mining

Various special materialisation and processing strategies have been developed for specific kinds of query patterns.

These include sampling cubes, ranking cubes, prediction cubes, multifeature cubes, and exception-based cubes.

These have a more automated data-mining role, along with attribute-oriented induction that finds compact descriptions of data instances.

Collectively, these kind of processing strategies are called **exploratory multidimensional data mining** or **online analytical data mining (OLAM)**.

5.1. Ranking Cubes: Top-k queries (Text:5.3.2)

Top-k queries are common for searching in web databases, k-nearest-neighbour for recommender systems, and similarity queries in multimedia databases. They enable searches for data that is *similar* to some particular data item, where similarity is defined by the application. The best *k* results, for some application-specified *k*, are returned in ranked order.

Example

Search for apartments with expected price 1000 and expected square metres 800

Select top 3 from Apartment
where City = "Canberra" and Num_Bedroom >= 2
order by [price – 1000]^2 + [sq metre - 800]^2 asc

Uses

- **selection condition** e.g.*City = "Canberra" and Num_Bedroom >= 2*
- **ranking function** e.g. *[price – 1000]^2 + [sq metre - 800]^2* (N.B L_2 norm!)

Results may be returned as a cube for OLAP analysis.

Queries are answered in a partial materialisation approach over expected pre-materialised selection dimensions (typically the 1-D cuboids for expected selection dimensions). A **ranking cube** with count measures assists in locating the top *k* for *k* given at run-time. Selection and ranking of the query are processed together.

The method achieves low space overhead and high performance in answering ranking queries with a high number of selection dimensions

5.2. Discovery-Driven Exploration (Text: 5.4.3)

A **Discovery-driven approach** to data cube exploration uses data mining to assist the user's analysis process.

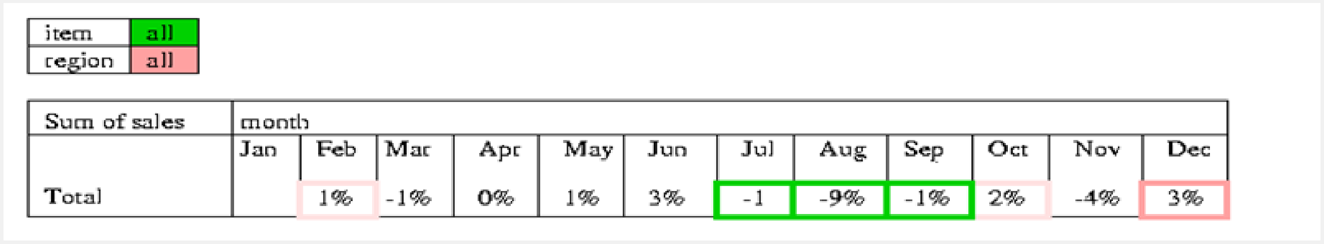
- Precomputed measures, i.e., **exception indicators**, mark regions that are significantly different from the normal values, at all levels of aggregation.
- Visual cues are used to highlight to user, e.g. colour shading indicating degree of unusualness.
- Computation of exceptions is interleaved with cube materialisation, for efficient algorithms. Exceptions are measures in the cells.

Exceptions can be:

- **SelfExp**: surprise of cell relative to other cells at same level of aggregation
- **InExp**: surprise beneath the cell for drill-down (accumulated over all drill-down dimensions) .
- **PathExp**: surprise beneath cell for each drill-down dimension.

<u>Example:</u>

Consider the cuboid below, being monthly change in sales (monthly difference from previous month) aggregated over item and region dimensions. Cell shading indicates degree of *SelfExp* (see this better in lower figures). The dark thick box borders for July, August and September in the cells below show high *InExp* values and prompt user to drill down. By selecting any cell you can see the degree of surprise by each path below that cell (*PathExp*), represented in the amount of shading of the dimension key at the top.

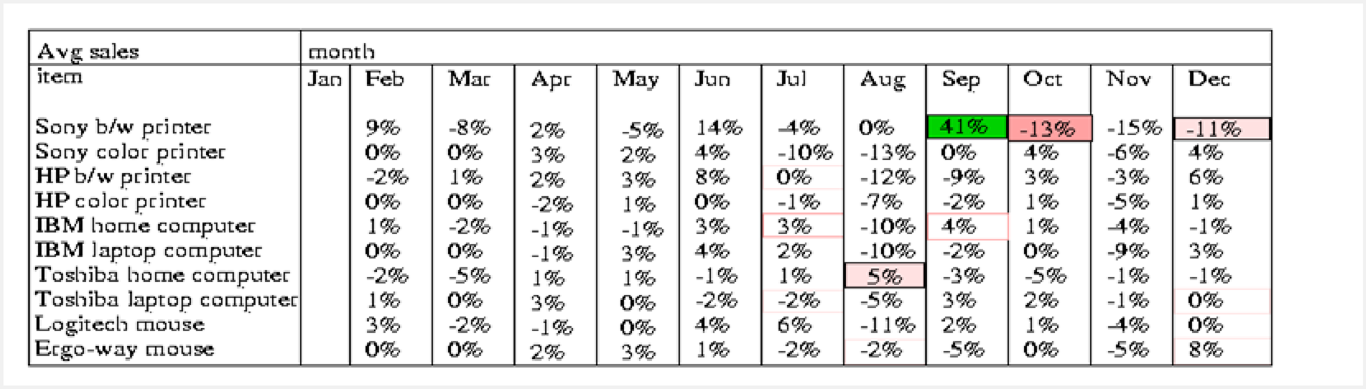


item	all
region	all

Sum of sales	month											
	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Total		1%	-1%	0%	1%	3%	-1	-9%	-1%	2%	-4%	3%

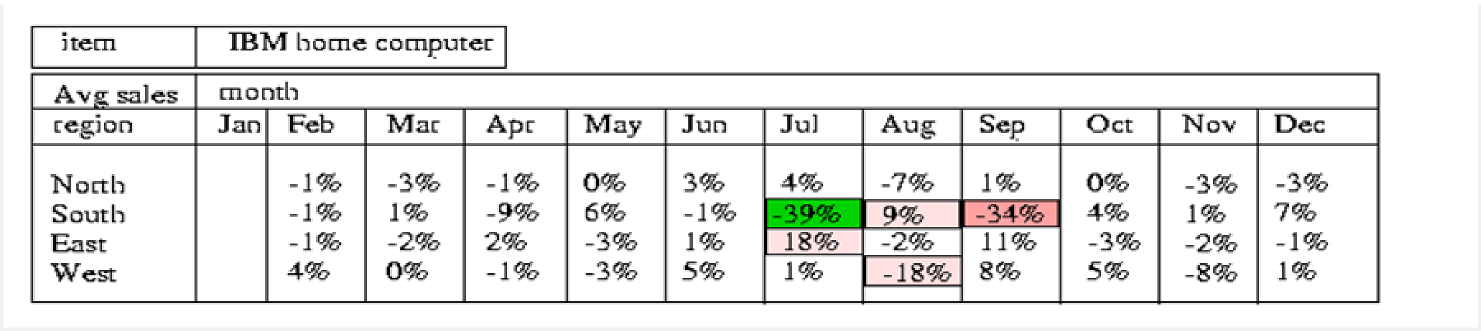
Driling down by *item* shows the cuboid below. *Sony b/w printer* in *September* has dark background shading indicating high *SelfExp.*Note that *Sony b/w printer* in *November* is not shaded, because although it appears exceptional across the rows, it is on-trend for its column (and accounting for both, it does not deviate much from its expected value and so is not particularly exceptional).
OTOH, *Sony b/w printer December i*s exceptional wrt its row and its column.

IBM home computers below has high *InExp* for *July* and *September,* shown by the border. Drilling down on *region* is shown in the next figure.



Avg sales	month											
item	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Sony b/w printer		9%	-8%	2%	-5%	14%	-4%	0%	41%	-13%	-15%	-11%
Sony color printer		0%	0%	3%	2%	4%	-10%	-13%	0%	4%	-6%	4%
HP b/w printer		-2%	1%	2%	3%	8%	0%	-12%	-9%	3%	-3%	6%
HP color printer		0%	0%	-2%	1%	0%	-1%	-7%	-2%	1%	-5%	1%
IBM home computer		1%	-2%	-1%	-1%	3%	3%	-10%	4%	1%	-4%	-1%
IBM laptop computer		0%	0%	-1%	3%	4%	2%	-10%	-2%	0%	-9%	3%
Toshiba home computer		-2%	-5%	1%	1%	-1%	1%	5%	-3%	-5%	-1%	-1%
Toshiba laptop computer		1%	0%	3%	0%	-2%	-2%	-5%	3%	2%	-1%	0%
Logitech mouse		3%	-2%	-1%	0%	4%	6%	-11%	2%	1%	-4%	0%
Ergo-way mouse		0%	0%	2%	3%	1%	-2%	-2%	-5%	0%	-5%	8%

The exception in sales difference on *region* is shown as dark shading (high *SelfExp*) below in the *Southern region* for *July* and *September.*



item	IBM home computer

Avg sales	month											
region	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
North		-1%	-3%	-1%	0%	3%	4%	-7%	1%	0%	-3%	-3%
South		-1%	1%	-9%	6%	-1%	-39%	9%	-34%	4%	1%	7%
East		-1%	-2%	2%	-3%	1%	18%	-2%	11%	-3%	-2%	-1%
West		4%	0%	-1%	-3%	5%	1%	-18%	8%	5%	-8%	1%

5.3. Attribute oriented induction (Text: 4.5)

Attribute oriented induction produces **concept descriptions** (also called **class descriptions**) of groups of data, using the language of the attributes and concept hierarchies around which the warehouse is designed.

Concept descriptions may be a **characterisation** of data in the class, or a **comparison** or **discrimination** comparing multiple classes.

The following is a simplified presentation for **characterisation.**

User first provides a relational database query that defines the class of data of interest.

Example: *select Name, University, location, Degree from CV where discipline = "computer science"*

Tuples in the answer are written as attribute-value pairs, with counts

Example: *Name = 'Kerry" and University = ANU and location = Canberra and Degree = MADA*, 1

The algorithm attempts to generalise the set of tuples to simplify their description.

How it is done?

1. Collect the task-relevant data (initial relation) using a relational database query
2. Perform generalization by attribute removal or attribute generalization (various heuristics may be used to determine how far to generalise)
    - Remove the attribute if it has a lot of different values, and so is not selective (e.g. Name)
    - Try replacing the attribute by another up a level in a concept hierarchy, and use that (eg. Degree -> Degreelevel)
3. Apply aggregation by merging identical, generalised tuples and accumulating their respective counts and other measures
    - generalisation will have collpased some tuples to identity so combine and aggregate measures.
4. Interaction with users for knowledge presentation or adjustment

Example: after induction, we have the following concept description of computer science CVs.

University	DegreeLevel	Count
UNSW	Undergrad	20
ANU	Postgrad	30
ANU	Undergrad	5

For **discrimination**, the method is similar, with separate starting queries to identify the target class and one or more discriminating classes. The discriminating classes serve as constraints on the generalisation of the target class to avoid including answers of the discriminating classes.

6. Reading

ACTION: Please read this paper on Mining in Cubes if you would like to deepen your understanding of OLAM (optional).

Ramakrishnan_and_Chen_2007

7. Practical Exercises

ACTION: These exercises continue on with the US population database you built previously. Have a go and hints and solutions are below.

Exercises for Analysing Data in Warehouses

Solution to Exercises: Analysing Data in Warehouses

8. Quiz

ACTION: blended only Please do the weekly quiz now.

ACTION: online or hybrid or on campus You should attempt the weekly quiz before your lab this week.

Quiz: analysing data in warehouses