## Views

```
create or replace view curr_dept_emp(emp_no, dept_no) as
    select emp_no, dept_no
    from dept_emp
    where from_date <= current_date and to_date >= current_date;

select * from curr_dept_emp limit 10;
```

To obtain info from a view, do a normal query as if it was a table.

It is possible to have views over views. Just need to address the desired views when creating the new one.

```
create or replace view count_emp_dept(dept_no, count_emp) as
    select dept_no, count(emp_no) as count_emp
    from curr_dept_emp
    group by dept_no;
```

**Query Unfolding**:

- Process of replacing and expanding a query expression with the expressions for the views in that query.
- It can be repeated until the query is fully expanded into an expression that no longer refers to views.
- Should be applied to verify if the expression is satisfiable and to possibly optimize its execution.

**When is a view relevant to a query?**

- View shares one or more subgoals with the query.
- Set of relations it mentions overlaps that of the query.
- Query applies predicates to attributes it has in common with the view. In this case, the view must apply either equivalent or logically weaker predicates to be part of an **equivalent rewriting**.
- When the view applies a locally stronger predicate, it may be part of a **contained rewriting**.

**Global-as-View (GAV)**:

- The global schema is defined as views over the local schema (data sources).
- **PRO** – The result of the query is easily calculated.
- **CONS**:
    - The reformulation may not be the most efficient method to answer the query.
    - Some subgoals are redundant.
    - Adding and removing sources involves considerable work and knowledge of the sources. -> Potentially not scalable.

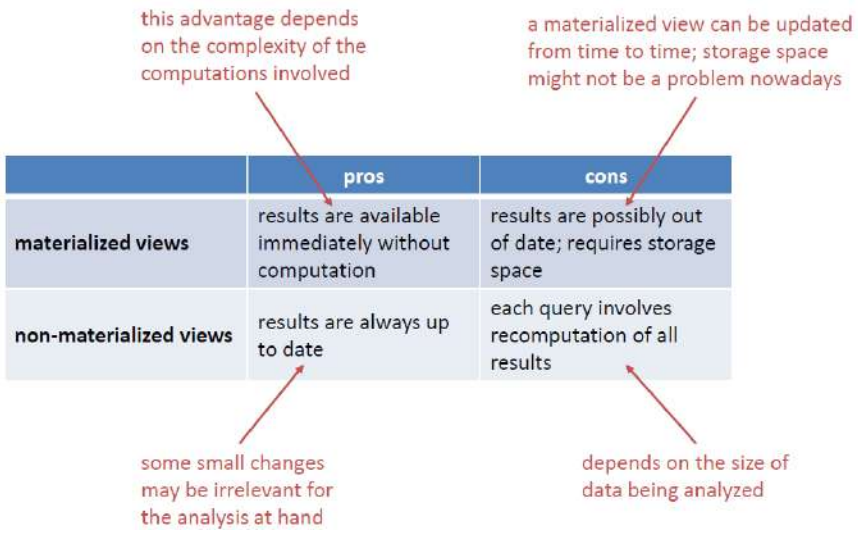
**Local-as-View (LAV)**:

- Local schemas defined as views over the global schema.
- Local sources are views over the mediated schema.
- Sources have the data | mediated schema is virtual.
- System must use the sources (views) to answer queries over the mediated schema.
- To answer a query formulated over the mediated schema, we need to reformulate it into a query over the known views (data sources).
- **PROS**:
    - Expresses incomplete information.
    - Mediated schema is clearly defined -> less likely to change.
    - Sources can be more accurately described.
- **CON** – Computationally expensive.

## Materialized vs Non-Materialized Views

**Materialized View** - The results are computed once and then persistently stored for later use.

**Non-Materialized View** - The results are computed every time the view is queried (default).




# Data Integration

**Schema** – Data source. Not necessarily a relational database.

**Schema Matching:**

- Find correspondence between elements of a source schema and elements of a target schema (elements can be relations or attribute names).
- It is mandatory to have the correspondence between elements in order to know which data transformation (query expressions) can be applied to the data in the source to obtain the data in the target.
- Can be done using 2 techniques:
    1. Apply string matching functions to the column names in order to find similar ones.
    2. Take into account the values of instances and infer whether the corresponding elements store information about the same subject.
- Machine learning is useful to avoid repetitive schema matching performed by humans. The system learns from previous schema matches and is able to predict the schema matching for a new source.

**Mediated Schema** – Schema resulting from a subset of attributes from different data source schemas.

**Wrappers** – Applied to data sources. Facilitate and simplify access to them.

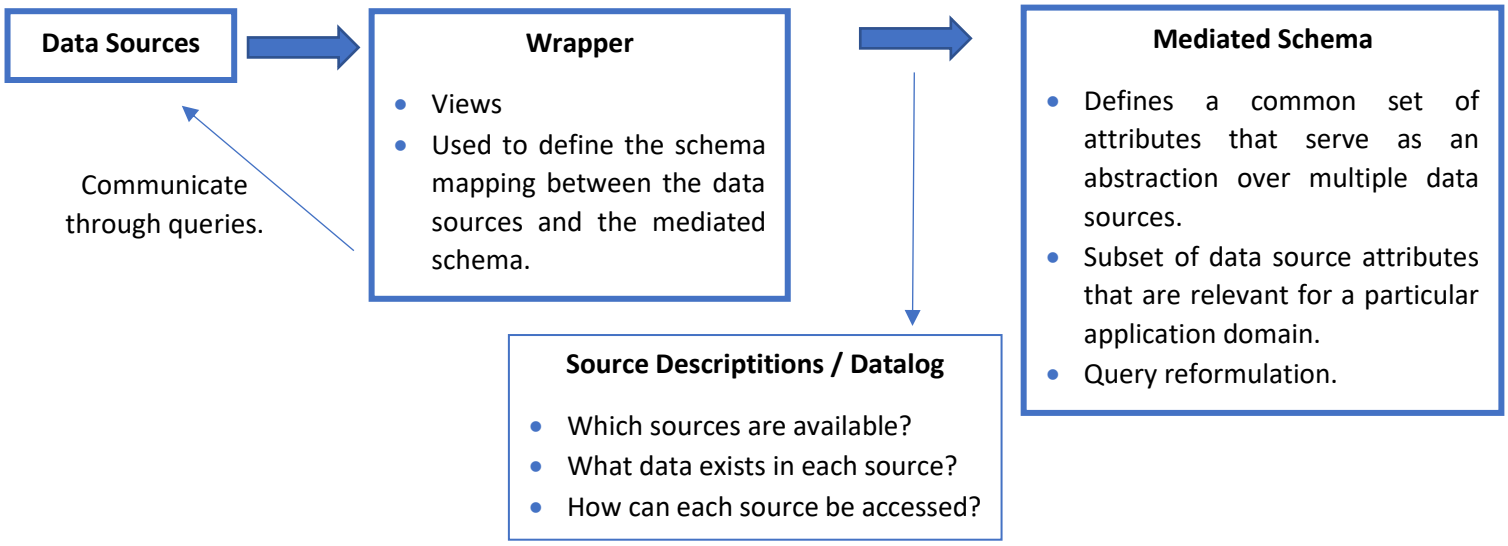
**Schema Mapping** – Is done from data sources to a mediated schema. Corresponds to the queries that bring data from local schema to a global mediated schema.

**Data Matching** – Find exact / approximate duplicates from different sources that may need to be merged, converted, or duplicated.

## View Materialization

Working with **non-materialized views**:

1. Define mediated schema.
2. Define the schema mappings between the data sources (or their wrappers) and the mediated schema,
3. Write query over mediated schema.
4. Use query-unfolding to reformulate query over the mediated schema into a query over the data sources.
5. Results are computed on the fly. It may take some time, but results are always up to date.




Working with **materialized views**:

1. Design a data warehouse.
   - A data warehouse has a certain schema, which plays a similar role to the mediated schema.
2. Implement an extract-transform-load (ETL) process to bring the data from the data sources into the warehouse.
   - **Extract** – Similar to using wrappers.
   - **Transform** – Similar to schema mapping.
   - **Load** – Materialized the result.
3. Retrieve pre-computed results.
   - It will be fast, but results may be outdated.
   - Re-run ETL process to update data warehouse.
4. Results might need to be aggregated.
   - Aggregate multiple dimensions.
   - Data warehouse is optimized for such aggregations.

An **ETL process** is composed of multiple **transformations**, which:

- Work in 'streaming mode', meaning the first row could reach the output before the second row is read from input, unless some aggregation on multiple rows needs to be performed.
- Can be saved and executed multiple times -> every time the materialized view needs to be updated.
- Can become quite complex.

As a rule, filtering rows, selecting columns and especially table joins should be performed by the database system, **unless** the requires data comes from multiple data sources, or when processing is difficult to implement in SQL, as is the case of duplicate detection with string matching, for instance.

**Why use an ETL tool?**

- Data comes from different sources.
- Data sources other than databases (text file, for instance).
- Complex data merging and transformations.
- Approximate matching, duplicate detection, data cleaning.
- Materialization into different output (databases, files, etc).

## String Matching

**String matching** in Data Integration can be **useful to**:

- Match data.
- Match schemas.
- Match schemas via data matching.

As such, string matching is based in **measures of distance or similarity**, which can be:

1. **Sequence-based** – Compute the cost of transforming one string into the other:
   - Edit / Levenshtein distance → Compare emails. Can align characters such as @.
   - Demerau-Levenshtein distance
   - Needleman-Wunsch distance
   - Jaro similarity measure → Compare 1st names.
   - Jaro-Winkler similarity measure →
2. **Set-based** – View the strings as sets or multi-sets of tokens, and use set-related properties to compute similarity scores:
   - Overlap similarity measure
   - Jaccard similarity measure → Compare phone numbers. Transpositions are useful.
   - TF/IDF similarity measure
3. **Hybrid** – Combine the benefits of sequence and set-based methods:
   - Generalized Jaccard, which enables approximate matching between tokens.
4. **Phonetic** – Match strings based on their sound:
   - Soundex → Compare last names.
   - Refined Soundex

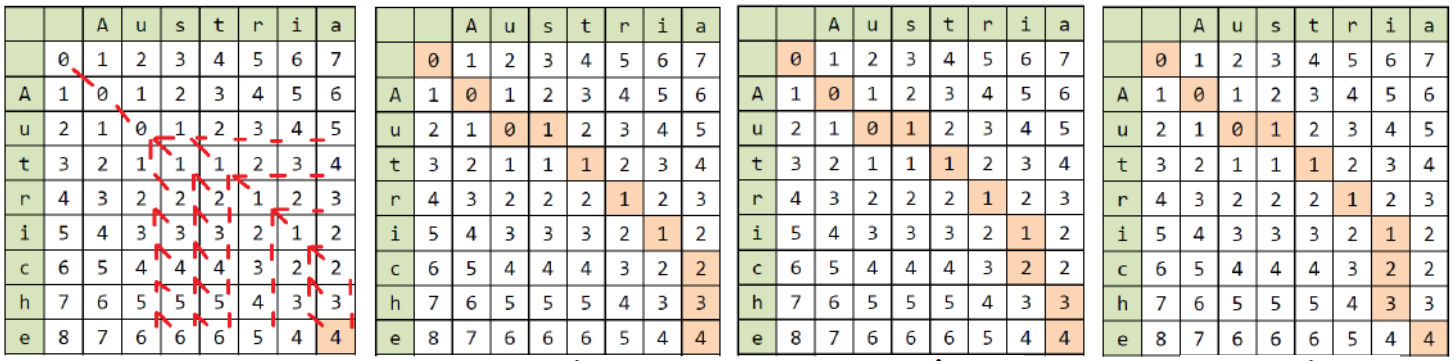
Any distance measure can be converted into a similarity measure.

$$s(x,y) = 1 - \frac{d(x,y)}{\max\{|x|,|y|\}}$$

## Edit / Levenshtein Distance

Counts the number of transformations (mismatch + use gap) to transform one string into another. The higher the count, the more distant the string are to one another.

The minimum number of changes needed corresponds to an optimal alignment of the strings.



| | | A | u | s | t | r | i | a |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| A | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| u | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 |
| t | 3 | 2 | 1 | 1 | 1 | 2 | 3 | 4 |
| r | 4 | 3 | 2 | 2 | 2 | 1 | 2 | 3 |
| i | 5 | 4 | 3 | 3 | 3 | 2 | 1 | 2 |
| c | 6 | 5 | 4 | 4 | 4 | 3 | 2 | 2 |
| h | 7 | 6 | 5 | 5 | 5 | 4 | 3 | 3 |
| e | 8 | 7 | 6 | 6 | 6 | 5 | 4 | 4 |

Austria__
Au_triche

| | | A | u | s | t | r | i | a |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| A | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| u | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 |
| t | 3 | 2 | 1 | 1 | 1 | 2 | 3 | 4 |
| r | 4 | 3 | 2 | 2 | 2 | 1 | 2 | 3 |
| i | 5 | 4 | 3 | 3 | 3 | 2 | 1 | 2 |
| c | 6 | 5 | 4 | 4 | 4 | 3 | 2 | 2 |
| h | 7 | 6 | 5 | 5 | 5 | 4 | 3 | 3 |
| e | 8 | 7 | 6 | 6 | 6 | 5 | 4 | 4 |

Austri_a_
Au_triche

| | | A | u | s | t | r | i | a |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| A | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| u | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 |
| t | 3 | 2 | 1 | 1 | 1 | 2 | 3 | 4 |
| r | 4 | 3 | 2 | 2 | 2 | 1 | 2 | 3 |
| i | 5 | 4 | 3 | 3 | 3 | 2 | 1 | 2 |
| c | 6 | 5 | 4 | 4 | 4 | 3 | 2 | 2 |
| h | 7 | 6 | 5 | 5 | 5 | 4 | 3 | 3 |
| e | 8 | 7 | 6 | 6 | 6 | 5 | 4 | 4 |

Austri__a
Au_triche

Given 2 strings $x = x_1x_2...x_n$ and $y = y_1y_2...y_m$, use the following recurrence equation to calculate the edit distance between x and y.

$$d(i,j) = min \begin{cases} \begin{cases} d(i-1,j-1) & if \quad x_i = y_j \\ d(i-1,j-1)+1 & if \quad x_i \neq y_j \end{cases} & \text{(diagonally)} \\ d(i-1,j)+1 & \text{(vertically)} \\ d(i,j-1)+1 & \text{(horizontally)} \end{cases}$$

In the presence of **very short / long strings**, **edit distance is not a good measure by itself**:

- For short strings, the distance would always be small, even if the strings were completely different.
- For very long strings, the distance can be large even if the strings are mostly similar.

The length of the strings should be taken into account.

**How to compare two addresses**:

1. We could not apply directly any of the string-matching algorithms.
2. We should first split each string into different fields.
3. Then, we should normalize each of these fields using a dictionary of abbreviations.
4. Finally, we should match the fields using string matching functions.

## Demerau-Levenshtein Distance

It is like the edit distance, but considers one more change:

- Mismatch
- Use Gap
- **Transposition** between 2 adjacent characters. In the edit distance would correspond to 2 changes.

## Needleman-Wunsch Similarity Measure

It is like the edit distance except mismatch and use gap have negative score and a match has a 0 or positive score. It is important to note that scores can be adjusted as on a per-letter basis, which requires the definition of a **score matrix**. Moreover, being a similarity measure, the higher the score, the more alike the strings are.

mismatch = -1 gap = -1 match = 0

| | | A | u | s | t | r | i | a |
|---|---|---|---|---|---|---|---|---|
| | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 |
| A | -1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 |
| u | -2 | -1 | 0 | -1 | -2 | -3 | -4 | -5 |
| t | -3 | -2 | -1 | -1 | -1 | -2 | -3 | -4 |
| r | -4 | -3 | -2 | -2 | -2 | -1 | -2 | -3 |
| i | -5 | -4 | -3 | -3 | -3 | -2 | -1 | -2 |
| c | -6 | -5 | -4 | -4 | -4 | -3 | -2 | -2 |
| h | -7 | -6 | -5 | -5 | -5 | -4 | -3 | -3 |
| e | -8 | -7 | -6 | -6 | -6 | -5 | -4 | -4 |

Example of Score Matrix:

| | a | b | c | d | e | f | … |
|---|---|---|---|---|---|---|---|
| a | +2 | -2 | -2 | -2 | -1 | -2 | … |
| b | -2 | +1 | -1 | -1 | -2 | -1 | … |
| c | -2 | -1 | +1 | -1 | -2 | -1 | … |
| d | -2 | -1 | -1 | +1 | -2 | -1 | … |
| e | -1 | -2 | -2 | -2 | +2 | -2 | … |
| f | -2 | -1 | -1 | -1 | -2 | +1 | … |
| … | … | … | … | … | … | … | … |

from score matrix

$$s(i,j) = max \begin{cases} s(i-1,j-1) + c(x_i, y_j) \\ s(i-1,j) + c_g \\ s(i,j-1) + c_g \end{cases}$$

note: max

gap score (negative)

It is used to compare short strings (first / last names). Its focus is on common characters and transpositions:

- **Common characters** $x_i$ and $y_j$ must be equal and not too distance => $x_i = y_j \wedge |i-j| \leq \frac{\min\{|x|,|y|\}}{2}$.
- **Sequences of common characters** should be equal unless there are transpositions.
- Each **mismatch counts as 1 transposition**.

Austria
Autriche
$c = 5$
$t = 0$
$jaro \approx 0.78$

Ireland
Ierland
$c = 7$
$t = 2$
$jaro \approx 0.95$

$$jaro(x,y) = \frac{1}{3}\left(\frac{c}{|x|} + \frac{c}{|y|} + \frac{c - \frac{t}{2}}{c}\right)$$

## Jaro-Winkler Similarity Measure

It is a variant of Jaro for strings with a common prefix.

Let PL be the length of the prefix, and PW its weight (usually 0.1), then:

$$jarowinkler(x,y) = (1 - PL * PW) * jaro(x,y) + PL * PW$$

## Jaccard Similarity Measure

It is a set-based measure based on n-grams (usually bigrams). **Remember** -> Sets do not have duplicates!

Austria {#A, Au, us, st, tr, ri, ia, a#}

Autriche {#A, Au, ut, tr, ri, ic, ch, he, e#}

$jaccard(x,y) = \frac{4}{13} \approx 0.31$

$$jaccard(x,y) = \frac{|B_x \cap B_y|}{|B_x \cup B_y|}$$

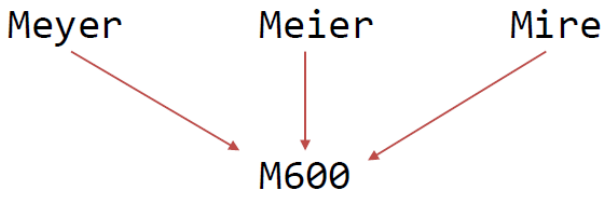
number of common bigrams (no duplicates)

number of all bigrams (no duplicates)

**Soundex Measure**

Phonetic measure used primarily to match surnames. It is calculated following a set of rules:

1. Keep the 1st letter.
2. Remove all occurrences of H and W.
3. Replace each letter with the corresponding digit.
4. Collapse any sequence of identical digits.
5. Drop non-digit chars (except the 1st letter).
6. Keep only 4 characters (pad with 0s if needed).




**Properties** of the Soundex algorithm:

- It is tuned to English.
- Results in many false positives and some false negatives.

The **Refined Soundex Measure** differs of the original by having a table with more groups, including one for vowels and H, W and Y. The first letter is kept and encoded as well, and there is no truncation to 4 chars. This results in less false positives.

## Data Matching

The objective is to match data from different sources. As such we follow the algorithm:

**for** each string $x \in X$ **do**
  **for** each string $y \in Y$ **do**
    **if** $s(x, y) \geq t$ **then** return $(x, y)$ as a matched pair
  **end for**
**end for**

Potential **problems of data matching:**

- Large number of comparisons may be needed.
- There might be false negatives as it is difficult to find a single perfect measure.
- There is no way to avoid some false positives or some false negatives as there may be no clear-cut threshold.

## Duplicate Detection

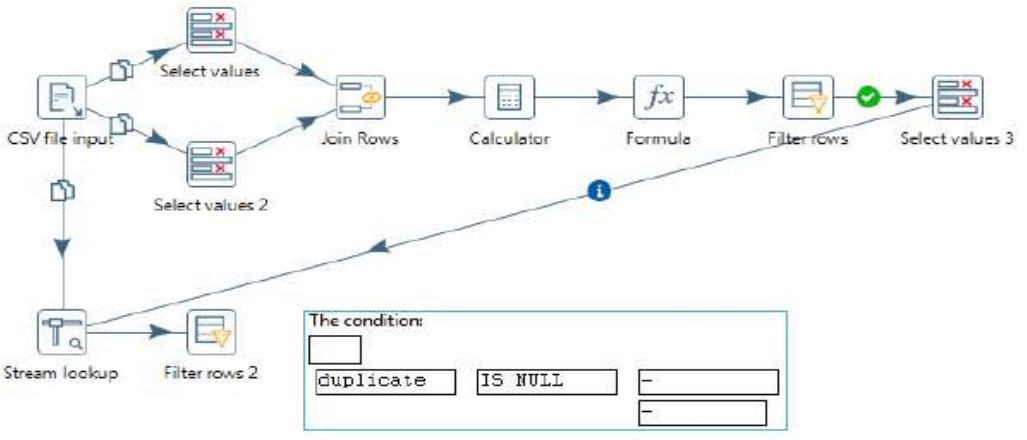
Like data matching, but instead of identifying records in X and Y that refer to the same real-world entity, we want to identify records in X that do so. These are called duplicates.

When we want to find exact duplicates, we test for equalities. However, when we want to find approximate duplicates, we use Similarity Measures.

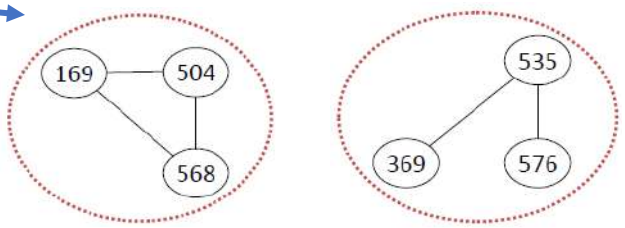
$$s(x, y) = \sum_{i=1}^{n} \alpha_i \cdot s_i(x, y) \geq t$$

Where x and y are records, $s_i(x, y)$ is the similarity between the $i^{th}$ attribute of x and y, and $\alpha_i$ its weight. $t$ is a pre-defined threshold.

Note that there is no need to compute the similarity between x and x, nor simultaneously the similarity between x and y and y and x. Therefore, **we only do so for records x and y where the $id_x < id_y$**.




There are **2 types of clusters of duplicates**.




**Transitivity** – If x and y are duplicates and y and z are duplicates, then x and z are (potential) duplicates too.

**Transitive Closure** – Set of all possible duplicates deduced using transitivity. The ones that match according to s(x, y) ≥ t (degree = 1) and the ones that can be deduced from those that matched (degree > 1).

After identifying the clusters, for each of them, we only want to keep one record. So, the question is: **How to decide which record o keep? What about taking parts from different records?**

One **solution** is to keep the most frequent value from each column within each cluster:

1. Group by cluster_id and desired column. Count records within group.
2. Sort by cluster_id and count, in descending order (so most frequent is on top).
3. Get first row for each cluster_id.

| cluster_id | customer_id | first_name | last_name | address | city | country |
|---|---|---|---|---|---|---|
| 1 | 169 | Isabel | de Castro | Estrada da saúde n. 58 | Lisboa | Portugal |
| 1 | 504 | Isabel | Castro | Estrada da saúde 58 | Lisbon | Portugal |
| 1 | 568 | Isabella | de Castro | Estrada da saúde 58 | Lisboa | Portugal |
| 2 | 369 | Manuel | Rodriguez | Jardim das Rosas 32 | Lisboa | Portugal |
| 2 | 535 | Manuel | Rodrigues | Jardim das Rosas n. 32 | Lisboa | Portugal |
| 2 | 576 | Emanuel | Rodrigues | Jardim das Rosas 32 | Lisbon | Portugal |

| cluster_id | first_name | count |
|---|---|---|
| 1 | Isabel | 2 |
| 1 | Isabella | 1 |
| 2 | Manuel | 2 |
| 2 | Emanuel | 1 |

| cluster_id | first_name | count |
|---|---|---|
| 1 | Isabel | 2 |
| 2 | Manuel | 2 |

In general, **duplicate detection and elimination has 3 phases**:

1. **Matching** – Join and calculate similarity.
2. **Clustering** – Apply transitive closure.
3. **Merging** – Collapse duplicates.

## Data Profiling

**What is it?**

- Analyze the contents of a data source.
- Gather statistics about such data.
- Identify data quality problems (missing / incomplete data, errors, etc).
- Understand the logic and relationships between data (unique values, keys, foreign keys, and other constraints).

**Completeness Analysis** – How many records have NULL in a certain column?

**Value Distribution** – What is the number of records per a column?

**Uniqueness Analysis** – Which columns are populated only with unique values? This is useful to identify primary keys.

**Referential Integrity** – Does every id in x appear in y? And does every id in y appear in x? If the answer is yes to the former and no to the latter, we can conclude that id in x is a foreign key to id in y.

**String Analysis, Data Type Discovery, Pattern Discovery, Soundex Frequency Histogram**, and many more data profiling tasks.

> It is possible to do data profiling using the **DataCleaner plug-in of PDI**.

# Data Warehousing

**Transitional Processing (OLTP)** – Basically performing CRUD operations over the data.

**Analytical Processing (OLAP)** – Selecting data using mostly group by's.

## Multi-Dimensional Model

In a multi-dimensional model, data can be analyzed according to different dimensions. It is possible to combine them.

**Analytical queries** over a database **allow**:

- A non-uniform treatment of dimensions. For instance, using special functions to handle the time dimension.
- The use of complex aggregations to calculate measures, such as sum, or averages, for example.
- All the previous to be running in parallel with transactional queries, which has a **negative performance impact**.

One **solution** is having a separate data warehouse to separate OLTP and OLAP operations. This allows the design of a more suitable schema for analytical queries => Star Schema. **To perform analytical queries over a star schema**:

1. Join fact table with desired dimension tables.
2. Group by some level of the dimensions.
3. Aggregate some measure in the fact table.

**How to build a data warehouse?**

- Use an ETL process to extract data from the original data source, transform data to fit a star schema, and load the data onto the data warehouse. This usually requires 1 transformation per dimension table + 1 transformation for the fact table + 1 job that runs all the transformations in the correct sequence (1st dimension tables, then fact table).

## OLAP Operations

Each dimension has a **hierarchy of levels**.

**Drill-Down** – Going from a higher to a lower level.

```
select country, sum(linetotal) as sales
from fact_order natural join dim_customer
group by country;
```

⇩

```
select city, sum(linetotal) as sales
from fact_order natural join dim_customer
group by city;
```

⚠

```
select country, sum(linetotal) as sales
from fact_order natural join dim_customer
group by country;
```

⇩

```
select country, city, sum(linetotal) as sales
from fact_order natural join dim_customer
group by country, city;
```

✓

**Roll-Up** – Going from a lower to a higher level.

```
select country, city, sum(linetotal) as sales
from fact_order natural join dim_customer
group by country, city;
```

⇩

```
select country, sum(linetotal) as sales
from fact_order natural join dim_customer
group by country;
```

**Slice** – Selecting a particular value of a dimension level.

```
select productline, year, sum(linetotal) as sales
from fact_order natural join dim_product natural join dim_time
group by productline, year;
```

⇩

```
select productline, year, sum(linetotal) as sales
from fact_order natural join dim_product natural join dim_time
group by productline, year
having year = 2003;
```

⚠

```
select productline, year, sum(linetotal) as sales
from fact_order natural join dim_product natural join dim_time
group by productline, year;
```

⇩

```
select productline, year, sum(linetotal) as sales
from fact_order natural join dim_product natural join dim_time
where year = 2003
group by productline, year;
```

✓

**Dice** – Applying multiple slicing conditions.

**Pivot** – Changing the order of dimensions.

**Typical OLAP operation**:

- Drill-down and roll-up between levels.
- Slice and dice to select particular values.
- Pivot to rotate dimensions.

A DW is a relational database that sores historical data in a convenient way for multi-dimensional data analysis using OLAP operations. As such, it stores **facts**, which **have associated measures**.

**Facts** – Define the possible analysis dimensions and can be grouped by dimensions.

**Measures** – Define the quantity that is being analyzed.

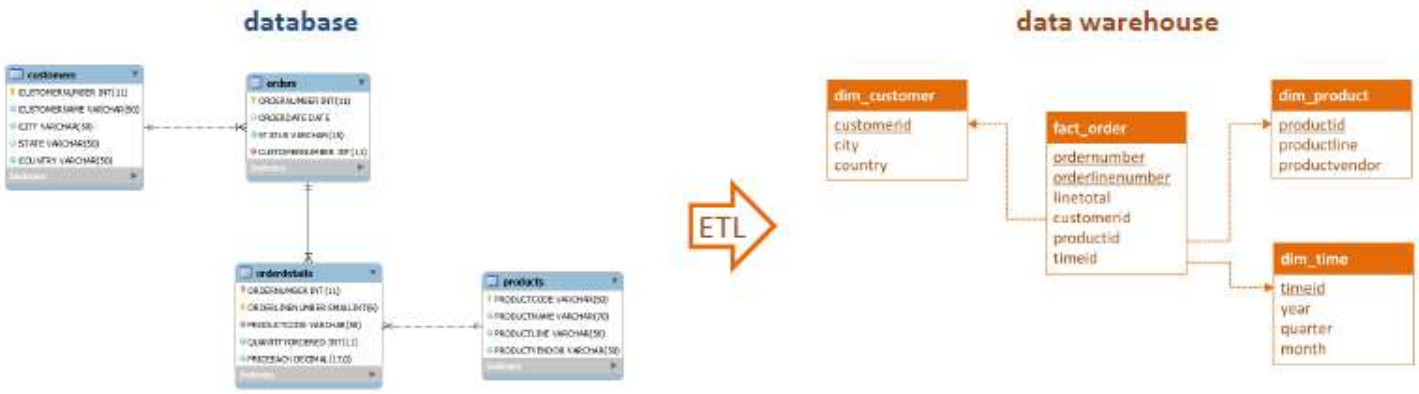
As facts are grouped by dimensions, their measures are aggregated at a certain dimension level.

In conclusion, **in a data warehouse**:

- Facts and measures are stored in a fact table.
- Each dimension is stored in a separate dimension level.
- The fact table has foreign keys to the dimension tables.
- Each dimension table stores its own dimension levels.

## ETL Process for a Data Warehouse

A **slowly changing dimension** keeps track of the multiple versions of the records that are changed over time. As such, instead of taking the initial record's id as the primary key, we need to create a **surrogate key**, which increments every time a record is added / changed in the database. To take note of the updates we need to re-run the transformation after the changes to the original database have been done.



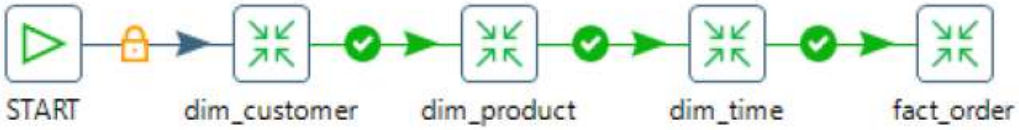


Basic Dimension


Slowly-changing Dimension


Time Dimension


Fact Table



## Data Warehouse Design

Defining a data warehouse implies thinking about 4 things:

1. **Schema** – Star, Snowflake, Starflake, Constellation.
2. **Hierarchies** – Balanced, Unbalanced, Recursive, Generalized, Ragged, Alternative, Parallel, Non-Strict.
3. **Measure** – Additive, Semi-Additive, Non-Additive, Derived.
4. **Time Dimension**

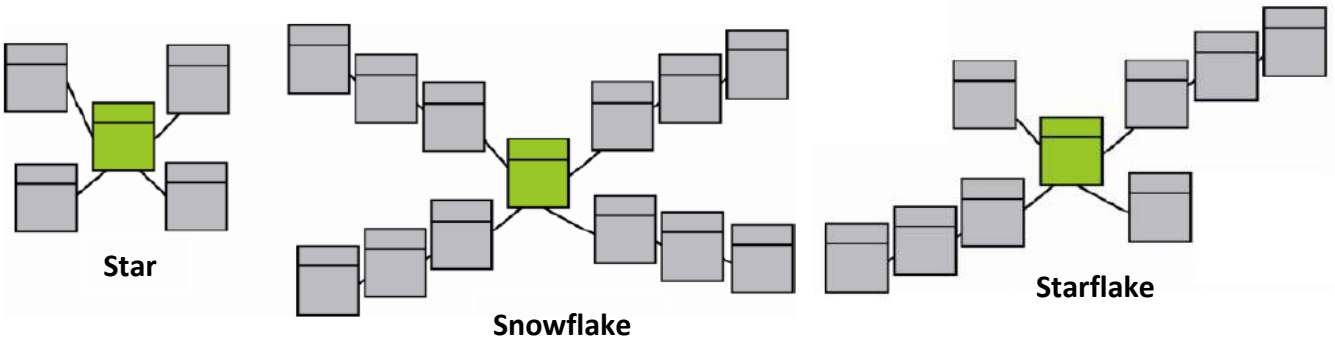
### Schemas

**Star Schema** – One table for each dimension, even in the presence of hierarchies. Denormalized dimension tables.

**Snowflake Schema** – Has normalized tables for all dimensions and their hierarchies.

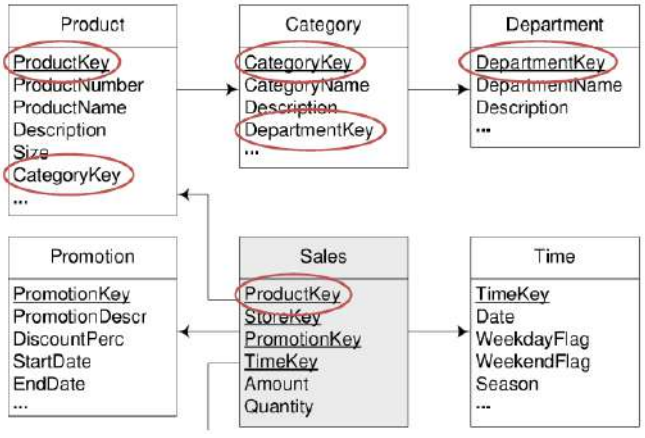
**Starflake Schema** – Mix between star and snowflake schemas, meaning not all dimension tables are normalized.

**Constellation Schema** – Has multiple fact tables, possibly with shared dimension tables, and is viewed as a collection of stars.




**How to normalize dimensions?**

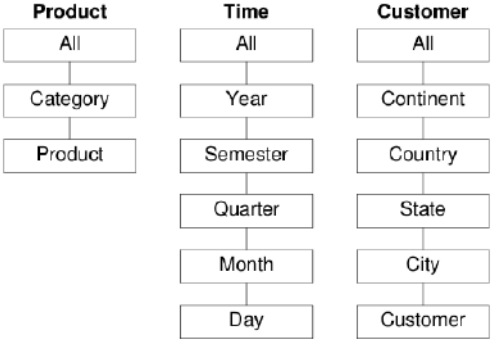
If a dimension stores redundant data (is denormalized), for instance categoryName and catgoryDesc, which are the same for multiple records, just replace them by categoryKey and move them to the Category table.




### Hierarchies

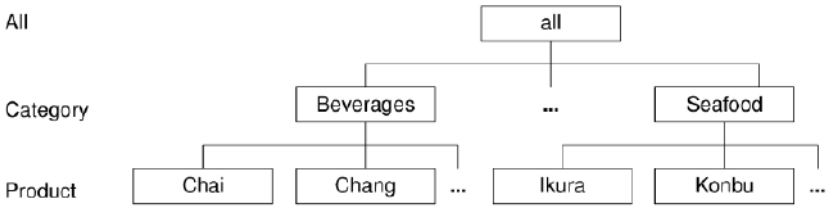
Dimension hierarchies are crucial to enable analysis at different levels of abstraction. The sequence of mappings relating lower-level concepts to higher-level ones is defined through the aggregation of measures.

In real-world applications, users must deal with complex hierarchies of various kinds. However, current OLAP systems support only a limited set of hierarchies.



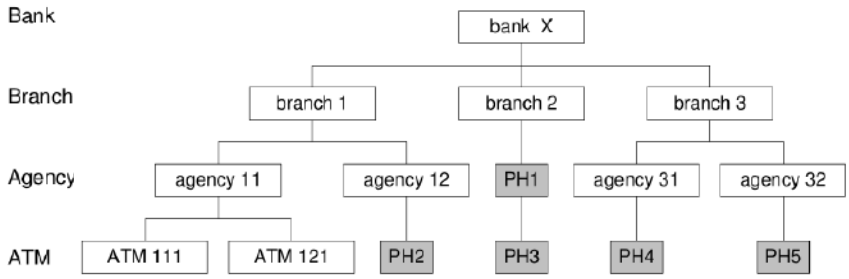
**Balanced**:

- All levels are mandatory.
- All branches have the same length.
- A child member has only one parent.
- Use flat table (as in star schema) or snowflake schema.



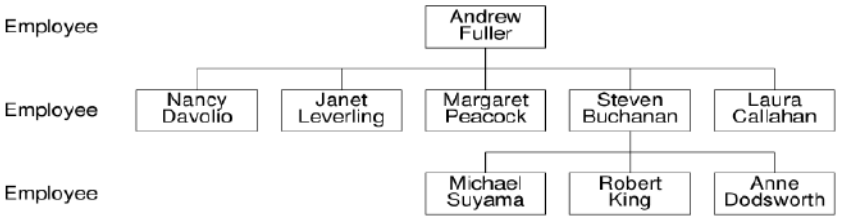

**Unbalanced**:

- Some levels are optional.
- Branches may have different lengths.
- A child has only one parent.
- Can be transformed into a balanced hierarchy by using placeholders.
- After balancing the hierarchy, you can use a star or snowflake schema.



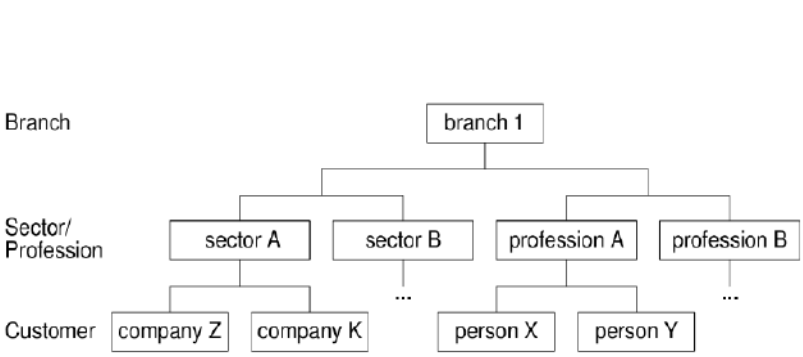

**Recursive**:

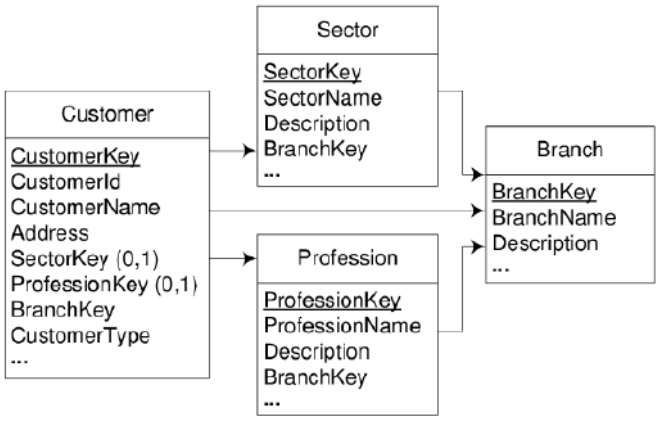
- All levels are of the same type.
- Can easily become unbalanced.
- Requires recursive queries to be traversed.




**Generalized**:

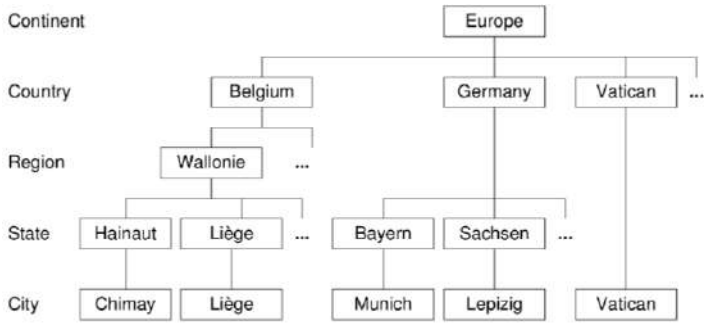
- The same level may have different types. For example, customer of a bank may be persons (with professions) or companies (belonging to a sector).
- Use flat table with NULLs or snowflake (preferred) with different aggregation paths for different types of customer.
- Add extra foreign key to roll-up to common level.



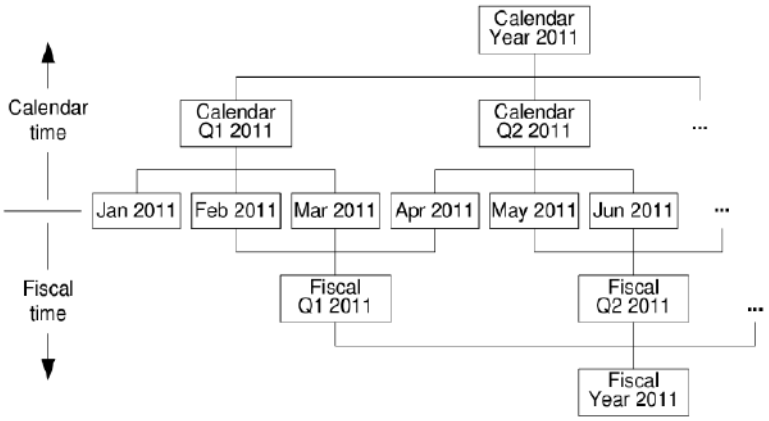

**Ragged**:

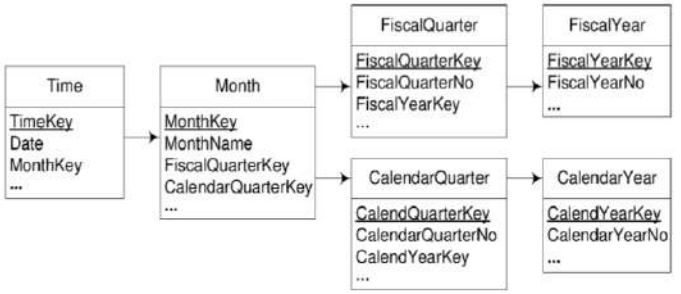
- One or more levels can be skipped.
- Several **solutions**:
  - Adding an extra foreign key to skip levels.
  - Using optional attributes for the levels to skip.
  - Using placeholders.



**Alternative**:

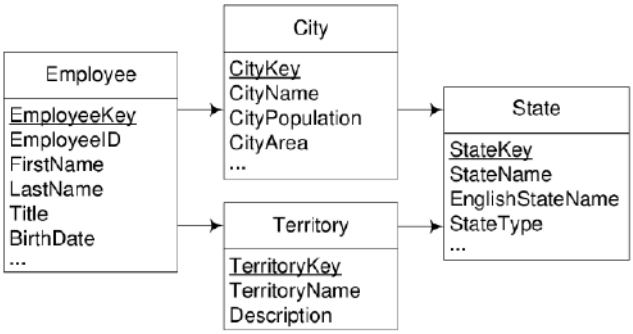
- The same level has alternative aggregation paths.
- Use snowflake schema.



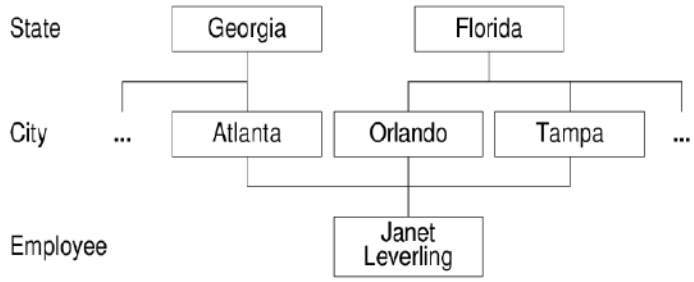

**Parallel**:

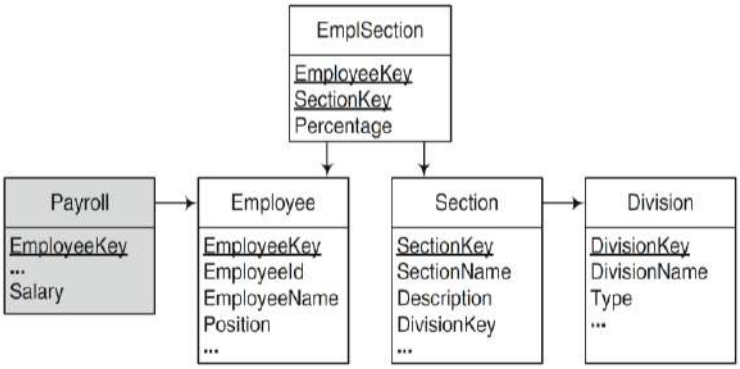
- The same dimension has several hierarchies associated with it.
- Use snowflake structure.




**Non-Strict**:

- A member may have several parents.
- Must take care to avoid problems of double count on roll-up.
- Use bridge table with percentage to distribute attribute.




## Measures

Each measure is associated to an aggregation function that combines several values into one. The **aggregation** takes place whenever we change to a different level in a dimension hierarchy.

When defining a measure, we must decide the associated aggregation function. *Sum* is the most typical, but it may not always apply. That is, some aggregation functions may not apply to a measure, or to a measure on a certain dimension.

**Additive** – Can be aggregated along all dimensions using addition.

**Semi-Additive** – Can be added along some, but not all, dimensions. (Inventory cannot be summed along time).

**Non-Additive** – Cannot be added along any dimension (Rates, for example).

**Derived** – Can be calculated from other measures or attributes.

## Time Dimension

A data warehouse is a historical database. Hence, the time dimension is almost always present.

In a **star/snowflake schema**, time is included both as a foreign key in the fact table and as a dimension containing the associated hierarchy levels.

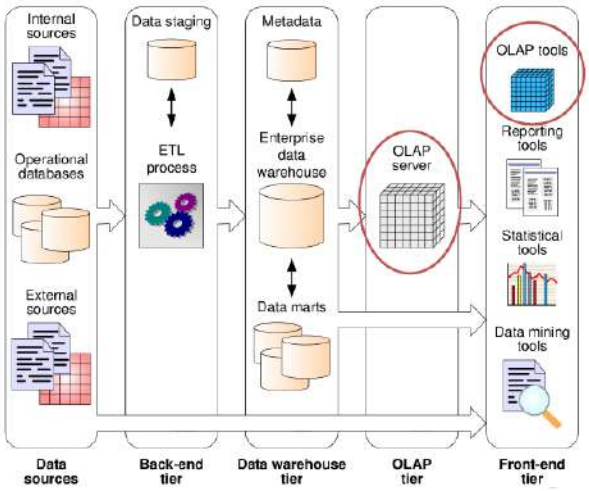
In **transactional databases**, time is stored in attributes of DATE type, and the different levels are computed on-the-fly using appropriate functions.

In **a DW**, time is stored explicitly in multiple attributes in the time dimension, making it easier to compute queries.

The time dimension:

- Has its granularity depending on use.
- May have more than 1 hierarchy (i.e. fiscal and calendar year).
- Can often be populated automatically.

## Data Warehouse Design (Pentaho Schema Workbench – PSW)




**OLAP Server**:

- Serves data (which may or may not come from a star schema) as a cube.
- Allows a client to run queries (MDX) over the data cube.
- Knows how to perform OLAP operations.
- Automatically calculates and aggregates measures.
- Needs a cube definition! -> XML file

```
-<Schema name="chinook_dw">
 -<Cube name="Invoices" visible="true" cache="true" enabled="true">
    <Table name="fact_invoice"> </Table>
  -<Dimension type="StandardDimension" visible="true" foreignKey="CustomerId" highCardinality="false" name="Customer">
    -<Hierarchy name="Customer Hierarchy" visible="true" hasAll="true" allMemberName="All Customers" primaryKey="CustomerId">
       <Table name="dim_customer"> </Table>
       <Level name="Country" visible="true" column="CustomerCountry" type="String" uniqueMembers="false" levelType="Regular" hideMemberIf="Never"> </Level>
       <Level name="City" visible="true" column="CustomerCity" type="String" uniqueMembers="false" levelType="Regular" hideMemberIf="Never"> </Level>
       <Level name="Customer Name" visible="true" column="CustomerName" type="String" uniqueMembers="false" levelType="Regular" hideMemberIf="Never"> </Level>
     </Hierarchy>
   </Dimension>
  -<Dimension type="StandardDimension" visible="true" foreignKey="TrackId" highCardinality="false" name="Tracks">
    -<Hierarchy name="Track Hierarchy" visible="true" hasAll="true" allMemberName="All Tracks" primaryKey="TrackId">
       <Table name="dim_track"> </Table>
       <Level name="Artist Name" visible="true" column="ArtistName" type="String" uniqueMembers="false" levelType="Regular" hideMemberIf="Never"> </Level>
       <Level name="Album Title" visible="true" column="AlbumTitle" type="String" uniqueMembers="false" levelType="Regular" hideMemberIf="Never"> </Level>
       <Level name="Track Name" visible="true" column="TrackName" type="String" uniqueMembers="false" levelType="Regular" hideMemberIf="Never"> </Level>
     </Hierarchy>
   </Dimension>
  -<Dimension type="TimeDimension" visible="true" foreignKey="TimeId" highCardinality="false" name="Time">
    -<Hierarchy name="Time Hierarchy" visible="true" hasAll="true" allMemberName="All Years" primaryKey="TimeId">
       <Table name="dim_time"> </Table>
       <Level name="Year" visible="true" column="YearId" type="Integer" uniqueMembers="false" levelType="TimeYears" hideMemberIf="Never"> </Level>
       <Level name="Month" visible="true" column="MonthName" ordinalColumn="MonthId" type="String" uniqueMembers="false" levelType="TimeMonths" hideMemberIf="Never"> </Level>
       <Level name="Day" visible="true" column="DayId" type="Integer" uniqueMembers="false" levelType="TimeDays" hideMemberIf="Never"> </Level>
     </Hierarchy>
   </Dimension>
   <Measure name="Sales" column="LineTotal" datatype="Numeric" formatString="$ #,###.00" aggregator="sum" visible="true"> </Measure>
   <Measure name="Quantity" column="Quantity" datatype="Integer" formatString="#,###" aggregator="sum" visible="true"> </Measure>
 </Cube>
</Schema>
```

## Slowly Changing Dimensions

A DW has its own primary keys, the **surrogate / technical keys**. These replace the original primary keys (natural keys) providing independence from keys in the original data sources. This is **important because** keys from multiple data sources may be inconsistent with each other and may change over time. Surrogate keys are represented as integers as to improve efficiency (avoid less efficient data types like strings).

**In a star schema**, each dimension is going to have its own surrogate key. **As for snowflake schemas**, both the dimensions and the levels have these keys.

There are **8 types of SCDs:**

- Type 0: retain original
- Type 1: overwrite
- Type 2: add row
- Type 3: add column
- Type 4: add mini-dimension
- Type 5: add mini-dimension and foreign key
- Type 6: extends type 2 with current value
- Type 7: add foreign key to fact table

Lets see each one of these types with the following example:

| TimeKey | EmployeeKey | CustomerKey | ProductKey | SalesAmount |
|---|---|---|---|---|
| t1 | e1 | c1 | p1 | 100 |
| t2 | e2 | c2 | p1 | 100 |
| t3 | e1 | c3 | p3 | 100 |
| t4 | e2 | c4 | p4 | 100 |

Sales fact table

| ProductKey | ProductName | CategoryName | Description |
|---|---|---|---|
| p1 | prod1 | cat1 | desc1 |
| p2 | prod2 | cat1 | desc1 |
| p3 | prod3 | cat2 | desc2 |
| p4 | prod4 | cat2 | desc2 |

Product dimension table

| EmployeeKey | CategoryName | SalesAmount |
|---|---|---|
| e1 | cat1 | 100 |
| e2 | cat1 | 100 |
| e1 | cat2 | 100 |
| e2 | cat2 | 100 |

Sales by employee and product category

Suppose prod1 changes from cat1 to cat2. We can't simply update its CategoryName because then it would show that e1 has a SalesAmount of 100 for prod1, with cat2. The problem is that e1 did those sales when prod1 was cat1!

**Solution with Type 0**: **retain original**

- Retain original value in the DW.
- Do not update with new value.
- Nothing changes in the product dimension table.

**Solution with Type 1**: **overwrite**

- Overwrite old value with new value.
- This solution assumes the modification is due to an error in the original data.
- History of the attribute is lost.

| ProductKey | ProductName | CategoryName | Description |
|---|---|---|---|
| p1 | prod1 | ~~cat1~~ cat2 | desc1 |
| p2 | prod2 | cat1 | desc1 |
| p3 | prod3 | cat2 | desc2 |
| p4 | prod4 | cat2 | desc2 |

**Solution with Type 2**: **add row**

- Store multiple versions of the same product.
- Add two columns with validity interval (from date, to date).
- This is where the surrogate key comes in handy.
- A product participates in the fact table with as many surrogates as there are attribute changes.

| Product Key | Product Name | Category Name | Description | From | To |
|---|---|---|---|---|---|
| p1 | prod1 | cat1 | desc1 | 2010-01-01 | 2011-12-31 |
| **p11** | prod1 | **cat2** | desc2 | 2012-01-01 | 9999-12-31 |
| p2 | prod2 | cat1 | desc1 | 2012-01-01 | 9999-12-31 |
| p3 | prod3 | cat2 | desc2 | 2012-01-01 | 9999-12-31 |
| p4 | prod4 | cat2 | desc2 | 2012-01-01 | 9999-12-31 |

- The number of distinct product keys no longer corresponds to the number of distinct products.
- There is another variant of Type 2 in which we add an extra row (which is redundant to the date ones) that says whether the row is Current or Expired.
- If using a **snowflak**e structure, changes in the Category table must be propagated to the product table.

**Solution with Type 3**: **add-column**

- Additional column for each attribute that might change.
- Stores only the two most recent versions.

| Product Key | Product Name | Category Name | New Category | Description | New Description |
|---|---|---|---|---|---|
| p1 | prod1 | **cat1** | **cat2** | **desc1** | **desc2** |
| p2 | prod2 | cat1 | Null | desc1 | Null |
| p3 | prod3 | cat2 | Null | desc2 | Null |
| p4 | prod4 | cat2 | Null | desc2 | Null |

**Solution with Type 4**: **add mini-dimension**

- For attributes that change frequently, create a mini-dimension.
- One row for each unique combination of those attributes instead of one row per product.
- Add ProductFeaturesKey to fact table.
- If the sales ranking of product p1 goes up, subsequent sales will be entered with pf1.

| TimeKey | EmployeeKey | CustomerKey | ProductKey | SalesAmount | ProductFeaturesKey |
|---|---|---|---|---|---|
| t1 | e1 | c1 | **p1** | 100 | pf2 |
| t2 | e2 | c2 | **p1** | 100 | pf2 |
| t3 | e3 | c3 | p1 | 50 | pf1 |

| Product FeaturesKey | Sales Ranking | Price Range |
|---|---|---|
| pf1 | 1 | 1–100 |
| pf2 | 2 | 1–100 |
| ... | ... | ... |
| pf200 | 7 | 500–600 |

**Solution with Type 5**: **add mini-dimension and foreign key**

- Extension of type 4 with foreign key being added to the dimension table.
- Makes it easier to analyze the current features of a product, with a roll-up based on current product features, for example.
- CurrentProductFeaturesKey is a Type 1 attribute, meaning it must be overwritten when the product features change.

| Product Key | Product Name | CurrentProduct FeaturesKey |
|---|---|---|
| p1 | prod1 | pf1 |
| ... | ... | ... |

- However, the fact table includes ProductFeaturesKey at the time of the sales, which is possibly different from CurrentProductFeaturesKey in the dimension table.

**Solution with Type 6**: **extends type 2 with current value**

- Extension of Type 2 with additional column for current value.
- This makes it possible to group by the categories at the time of sale (CategoryKey) and by the current categories (CurrentCategoryKey).

**At time of sale.**

| Product Key | Product Name | Category Key | From | To | Current CategoryKey |
|---|---|---|---|---|---|
| p1 | prod1 | c1 | 2010-01-01 | 2011-12-31 | **c11** |
| **p11** | prod1 | **c11** | 2012-01-01 | 9999-12-31 | **c11** |
| p2 | prod2 | c1 | 2010-01-01 | 9999-12-31 | c1 |
| p3 | prod3 | c2 | 2010-01-01 | 9999-12-31 | c2 |
| p4 | prod4 | c2 | 2011-01-01 | 9999-12-31 | c2 |

**Solution with Type 7**: **add foreign key to fact table**

- Used if there are several attributes for which we need to support both current and historical perspectives.
- **Type 6** would require one additional column in the dimension table for each of those attributes.
- Instead, just store the natural key in the fact table.
- This way, we can use ProductKey for historical analysis, that is, based on product values when they were sold, and ProductName to analyze current values (for this, we use a view with the current values).
- The current values are obtained from the current version of each product => from_date ≤ current_date ≤ to_date.
- There is a variant of Type 7 that uses only the surrogate key. This **avoids two foreign keys** for product in the fact table by using a view with current category for each product, where the product is now identified by the surrogate key.

| TimeKey | EmployeeKey | CustomerKey | ProductKey | Product Name | SalesAmount |
|---|---|---|---|---|---|
| t1 | e1 | c1 | p1 | prod1 | 100 |
| t2 | e2 | c2 | p11 | prod1 | 100 |
| t3 | e1 | c3 | p3 | prod3 | 100 |
| t4 | e2 | c4 | p4 | prod4 | 100 |

| Product Name | Category Key |
|---|---|
| prod1 | c2 |
| prod2 | c1 |
| prod3 | c2 |
| prod4 | c2 |

| Product Key | Current CategoryKey |
|---|---|
| p1 | **c11** |
| **p11** | **c11** |
| p2 | c1 |
| p3 | c2 |
| p4 | c2 |

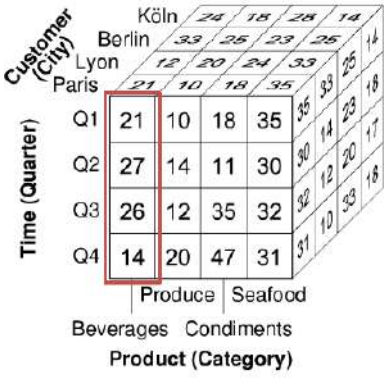
**Variant with surrogate key**

## MDX Queries

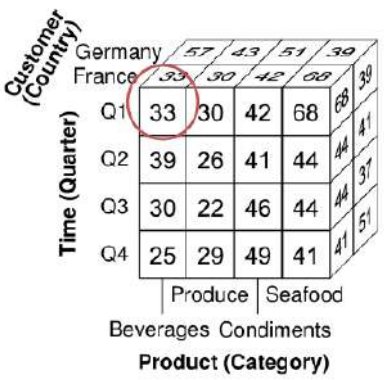
SQL is a language to query databases, since it operates over tables, columns, and records. As for MDX (Multi-Dimensional eXpressions), it is a language to query data warehouses. In fact, despite it being inspired in SQL (select…from…where), MDX queries operates over cubes, dimensions, and measures.

**Tuples** are of the form *(Dimension.Level.Member)* and sets of tuples are of the form *{(Dimension1.Level.Member), (Dimension2.Level.Member)}.* It is also possible to specify measures.

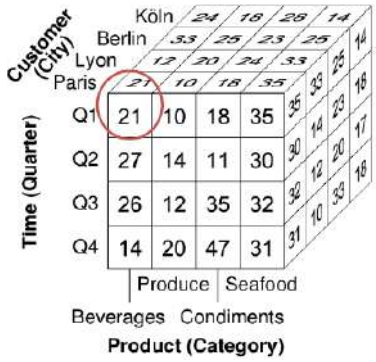
(Customer.City.Paris,
Product.Category.Beverages,
Time.Quarter.Members)




(Customer.Country.France,
Product.Category.Beverages,
Time.Quarter.Q1)




(Customer.City.Paris,
Product.Category.Beverages,
Time.Quarter.Q1,
Measures.SalesAmount)




A typical **MDX query** is of the form *SELECT <axis specification> FROM <cube> [WHERE <slicer specification>]*. (Axis(0) -> COLUMNS and Axis(1) -> ROWS).

It is possible to hide rows without values by using ***NON EMPTY*** before the corresponding *Dimension.Level.Member* on the *SELECT* clause. If a name has spaces in it, for instance 'Product Line' enclose it in [ ] -> *[Product Line]*. Specific values should also be enclosed -> *Time.[2003].[Q1]*.

Moreover, if a dimension appears in an axis (*SELECT*), it **cannot be used** in a slicer (*WHERE*), and vice-versa.

```
SELECT Measures.Members ON COLUMNS,
       Time.Year.Members ON ROWS
FROM Orders
WHERE {(Customer.Country.Italy,
        Product.[Product Line].[Classic Cars]),
       (Customer.Country.France,
        Product.[Product Line].[Classic Cars])}
```

| Year | Sales | Quantity |
|---|---|---|
| 2003 | $ 184,463.00 | 1,514 |
| 2004 | $ 274,296.00 | 2,382 |
| 2005 | $ 63,314.00 | 626 |

*Show all measures by year for customer in Italy or France who bought Classic Cars.*

```
SELECT Time.Year.Members ON COLUMNS,
       Customer.Country.Members ON ROWS
FROM Orders
WHERE (Measures.Sales,
       Product.[Product Line].[Classic Cars])
```

| Country | 2003 | 2004 | 2005 |
|---|---|---|---|
| Australia | $ 85,355.00 | $ 76,198.00 | $ 31,460.00 |
| Austria | $ 26,647.00 | $ 15,333.00 | $ 59,480.00 |
| Belgium | - | $ 3,510.00 | $ 16,631.00 |
| Canada | $ 28,896.00 | $ 20,600.00 | $ 12,138.00 |
| Denmark | $ 60,756.00 | $ 70,338.00 | $ 25,974.00 |

*Show sales by country and year for Classic Cars.*

If there is no ambiguity, all the following forms work:

- *Customer.[Customer Hierarchy].Country.France*
- *Customer.Country.France*
- *Customer.France*

Also note that the fully qualified form is *Dimension.Hierarchy.Level.Member*.

It is possible to use the ***CHILDREN*** member to select all the children of a level.

The use of ***DRILLDOWNLEVE****L* shows all the elements of the level immediately bellow the one provided as argument. It differs from the *CHILDREN* member because it also shows the 'parent' level.

```
SELECT Time.[2003].Children ON COLUMNS,
       {Customer.Country.Germany.Children,
        Customer.Country.Italy.Children} ON ROWS
FROM Orders
WHERE Measures.Sales
```

| City | Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|---|
| Frankfurt | $ 11,424.00 | - | - | $ 27,251.00 |
| Köln | - | - | - | $ 31,364.00 |
| Bergamo | $ 56,203.00 | - | - | $ 40,078.00 |
| Milan | - | - | - | $ 21,711.00 |
| Reggio Emilia | - | - | - | $ 44,648.00 |

*Sales by quarter of 2013 in German and Italian cities.*

```
SELECT Time.[2003].Children ON COLUMNS,
       {DRILLDOWNLEVEL(Customer.Country.Germany),
        DRILLDOWNLEVEL(Customer.Country.Italy)} ON ROWS
FROM Orders
WHERE Measures.Sales
```

| Country | City | Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|---|---|
| Germany | | $ 11,424.00 | - | - | $ 58,815.00 |
| | Frankfurt | $ 11,424.00 | - | - | $ 27,251.00 |
| | Köln | - | - | - | $ 31,364.00 |
| Italy | | $ 56,203.00 | - | - | $ 106,437.00 |
| | Bergamo | $ 56,203.00 | - | - | $ 40,078.00 |

*Sales by quarter in 2003 in German and Italian cities with country too.*

The use of ***DESDENDANTS*** may result in the same output as that of *CHILREN* if no third argument is provided or if it is equal to *SELF*. In fact, the third argument specifies which level (or levels) to include in the results. The 3rd argument can, then, be: *SELF, BEFORE, AFTER, SELF_AND_BEFORE, SELF_AND_AFTER, BEFORE_AND_AFTER, SELF_BEFORE_AFTER*.

***ASCENDANTS*** return a set that includes all the ancestors of a member and the member itself. ***ANCESTOR*** is applied to a specific level.

```
SELECT Time.[2003].Children ON COLUMNS,
       DESCENDANTS(Customer.Italy, Customer.City) ON ROWS
FROM Orders
WHERE Measures.Sales
```

| City | Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|---|
| Bergamo | $ 56,203.00 | - | - | $ 40,078.00 |
| Milan | - | - | - | $ 21,711.00 |
| Reggio Emilia | - | - | - | $ 44,648.00 |

```
SELECT Time.[2003].Children ON COLUMNS,
       DESCENDANTS(Customer.Italy,
                   Customer.City,
                   SELF_BEFORE_AFTER) ON ROWS
FROM Orders
WHERE Measures.Sales
```

| Country | City | Customer Name | Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|---|---|---|
| Italy | | | $ 56,203.00 | - | - | $ 106,437.00 |
| | Bergamo | | $ 56,203.00 | - | - | $ 40,078.00 |
| | | Rovelli Gifts | $ 56,203.00 | - | - | $ 40,078.00 |
| | Milan | | - | - | - | $ 21,711.00 |
| | | Frau da Collezione | - | - | - | $ 21,711.00 |

```
SELECT Time.[2003].Children ON COLUMNS,
       ASCENDANTS(Customer.City.Milan) ON ROWS
FROM Orders
WHERE Measures.Sales
```

| (All) | Country | City | Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|---|---|---|
| All Customers | Italy | Milan | - | - | - | $ 21,711.00 |
| | | | $ 56,203.00 | - | - | $ 106,437.00 |
| | | | $ 1,080,270.00 | $ 564,875.00 | $ 687,296.00 | $ 1,979,994.00 |

```
SELECT Time.[2003].Children ON COLUMNS,
       ANCESTOR(Customer.City.Milan,
                Customer.Country) ON ROWS
FROM Orders
WHERE Measures.Sales
```

| Country | Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|---|
| Italy | $ 56,203.00 | - | - | $ 106,437.00 |

***CROSSJOIN*** joins multiple dimensions on rows or columns.

```
SELECT Product.[Product Line].Members ON COLUMNS,
       CROSSJOIN(Customer.Country.Members,
                 Time.Year.Members) ON ROWS
FROM Orders
WHERE Measures.Sales
```

| Country | Year | Classic Cars | Motorcycles | Planes | Ships |
|---|---|---|---|---|---|
| Australia | 2003 | $ 85,355.00 | $ 42,337.00 | $ 22,316.00 | - |
| | 2004 | $ 76,198.00 | $ 33,077.00 | $ 41,416.00 | $ 1,080.00 |
| | 2005 | $ 31,460.00 | $ 14,478.00 | $ 11,086.00 | $ 3,072.00 |
| Austria | 2003 | $ 26,647.00 | - | $ 14,221.00 | $ 9,028.00 |
| | 2004 | $ 15,333.00 | $ 26,060.00 | $ 3,638.00 | - |

```
SELECT Product.[Product Line].Members ON COLUMNS,
       Customer.Country.Members * Time.Year.Members ON ROWS
FROM Orders
WHERE Measures.Sales
```

| Country | Year | Classic Cars | Motorcycles | Planes | Ships |
|---|---|---|---|---|---|
| Australia | 2003 | $ 85,355.00 | $ 42,337.00 | $ 22,316.00 | - |
| | 2004 | $ 76,198.00 | $ 33,077.00 | $ 41,416.00 | $ 1,080.00 |
| | 2005 | $ 31,460.00 | $ 14,478.00 | $ 11,086.00 | $ 3,072.00 |
| Austria | 2003 | $ 26,647.00 | - | $ 14,221.00 | $ 9,028.00 |
| | 2004 | $ 15,333.00 | $ 26,060.00 | $ 3,638.00 | - |

MDX also allows the definition of new members or measures inside the queries. It is also possible to **define an alias** for a set of members, as well as to define new measures based on relative navigation. All of this is possible with the use of ***WITH.***

```
WITH MEMBER Measures.SalesPerUnit AS
            (Measures.Sales / Measures.Quantity)
SELECT Measures.AllMembers ON COLUMNS,
       Customer.Country.Members ON ROWS
FROM Orders
```

| Country | Sales | Quantity | SalesPerUnit |
|---|---|---|---|
| Australia | $ 630,638.00 | 6,246 | $ 100.97 |
| Austria | $ 202,089.00 | 1,974 | $ 102.38 |
| Belgium | $ 108,485.00 | 1,074 | $ 101.01 |
| Canada | $ 224,085.00 | 2,293 | $ 97.73 |
| Denmark | $ 245,582.00 | 2,197 | $ 111.78 |

```
WITH SET TopCountries AS
   TOPCOUNT(Customer.Country.Members, 3, Measures.Sales)
SELECT Measures.Members ON COLUMNS,
       TopCountries ON ROWS
FROM Orders
```

| Country | Sales | Quantity |
|---|---|---|
| USA | $ 4,263,627.00 | 37,749 |
| Spain | $ 1,215,356.00 | 12,429 |
| France | $ 1,111,022.00 | 11,090 |

```
WITH MEMBER Measures.PercentSales AS
    (Measures.Sales, Customer.Country.CurrentMember)
    / (Measures.Sales, Customer.Country.CurrentMember.Parent),
    FORMAT_STRING = '#0.00%'
SELECT {Measures.Sales, Measures.PercentSales} ON COLUMNS,
       Customer.Country.Members ON ROWS
FROM Orders
```

| Country | Sales | PercentSales |
|---|---|---|
| Australia | $ 630,638.00 | 5.59% |
| Austria | $ 202,089.00 | 1.79% |
| Belgium | $ 108,485.00 | 0.96% |
| Canada | $ 224,085.00 | 1.99% |

```
WITH MEMBER Measures.[Previous Month] AS
            Time.Month.CurrentMember.PrevMember
     MEMBER Measures.[Sales Growth] AS
            Measures.Sales - Measures.[Previous Month]
SELECT {Measures.Sales,
        Measures.[Previous Month],
        Measures.[Sales Growth]} ON COLUMNS,
        DESCENDANTS(Time.Year.[2004], Time.Month) ON ROWS
FROM Orders
```

| Month | Sales | Previous Month | Sales Growth |
|---|---|---|---|
| Jan | $ 316,662.00 | 303,464 | $ 13,198.00 |
| Feb | $ 318,663.00 | 316,662 | $ 2,001.00 |
| Mar | $ 242,222.00 | 318,663 | -$ 76,441.00 |

It is possible to ***FILTER*** members based on a condition.

```
SELECT Time.Year.Members ON COLUMNS,
       FILTER(Customer.Country.Members,
              (Measures.Sales, Time.[2004]) > 250000) ON ROWS
FROM Orders
WHERE Measures.Sales
```

| Country | 2003 | 2004 | 2005 |
|---|---|---|---|
| France | $ 312,913.00 | $ 555,193.00 | $ 242,916.00 |
| New Zealand | $ 90,055.00 | $ 256,376.00 | $ 189,185.00 |
| Spain | $ 405,336.00 | $ 483,344.00 | $ 326,676.00 |
| UK | $ 180,317.00 | $ 257,648.00 | $ 40,755.00 |
| USA | $ 1,940,324.00 | $ 1,685,741.00 | $ 637,562.00 |

It is also possible to sort (***ORDER***) the results, as well as get a limited number of rows (***HEAD, TOPCOUNT, TOPPERCENT***).

```
SELECT Measures.Members ON COLUMNS,
       ORDER(Customer.Country.Members,
             Measures.Sales, DESC) ON ROWS
FROM Orders
```

```
SELECT Measures.Members ON COLUMNS,
       HEAD(ORDER(Customer.Country.Members,
                  Measures.Sales, DESC), 3) ON ROWS
FROM Orders
```

```
SELECT Measures.Members ON COLUMNS,
       TOPCOUNT(Customer.Country.Members, 3,
                Measures.Sales) ON ROWS
FROM Orders
```

```
SELECT Measures.Members ON COLUMNS,
       TOPPERCENT(Customer.Country.Members, 50,
                  Measures.Sales) ON ROWS
FROM Orders
```

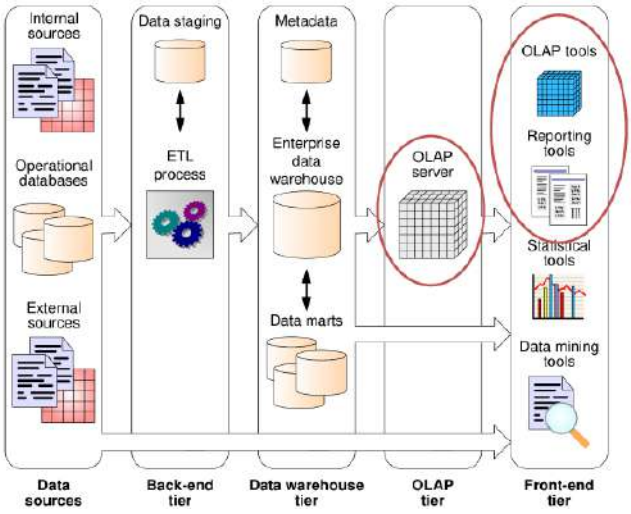
## Exploiting the Dara Warehouse

**OLAP tools**:

- Allow an explorative analysis of the DW.
- Facilitate the formulation of complex queries.
- Allow the user to build ad-hoc queries.

**Reporting tools**: (Pentaho Report Designer)

- Focus on paper-based or Web-based reports.
- Allow only fixed queries for specific information in a specific format.
- Useful when the same report / information is needed on a regular basis.




## Reports

Reports typically contain lists, tables, charts, key performance indicators (KPIs), among other. The data can come from some file, database, data warehouse, ERP system, website, and so on.

**Steps of using a reporting tool**:

1. **Input**:
   - Choose the data source (DB or DW).
   - Write the query (SQL or MDX).
   - Place the elements and graphics in the report -> Needs the most work to be done.
2. **Run**:
   - The report is generated automatically.
   - So just configure it once and run multiple times (whenever necessary).
3. **Output**:
   - Nicely-formatted report (PDF, HTML, etc)

Reporting is informative, but it is not enough. It provides no information on the results' trend, nor on whether the organization is achieving its goals.

## Key Performance Indicators - KPIs

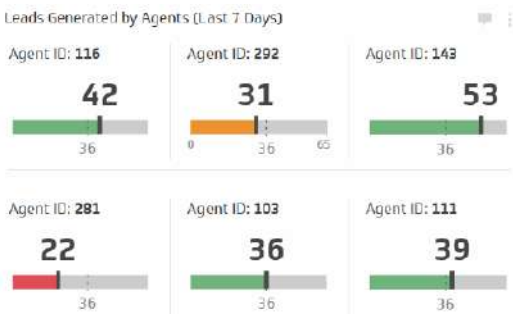
Since reports are not enough to evaluate business performance, we still need a way to do it. We need to be able to define business strategy and goals, as well as to compare actual results with goals or expectations.

As such, we use **KPIs**:

- Measurements to estimate the effectiveness of an organization.
- Have a current value, which is compared to a target value.
- Are typically included in reports and dashboards.

**Dashboard**:

- Visual display of important information in a single screen, so it can be monitored at a glance.
- Should be concise, clear, easy to interpret and self-explanatory.
- Should enable rapid visual identification of measures that are going away from goals (using color, for instance).



**How to define new KPIs?**

1. Identify relevant sources:
    - **Primary sources** – Employees, managers, boar, suppliers, and customers.
    - **Secondary Sources** – Strategic plan, business plan, annual reports, internal operational reports, competitor review reports, etc.
    - **External Sources** – Catalogues, websites, annual reports of other organizations, expert advice, questions in discussion forum, etc.
2. Categorize potential metrics according to perspective (e.g. Financial perspective, customer perspective, etc.).
3. For each metric:
    - Define the metric in precise terms.
    - Verify that it is aligned with business goals.
    - Check if there is reliable data to compute it.
    - Set its target value. Take into account historical information, industry benchmarks, economic conditions, etc.

## Technologies Used

**Pentaho Data Integration (PDI)**:

- Allows us to perform the ETL process and transform the data coming from the database into the new schema for the data warehouse.
- Useful to extract and transform data in a way we can analyze it.
- **Transformation** - Pipeline of processing steps that work in streaming mode.
- **Job** – Sequence of PDI transformations where each one is executed only after the previous ones have successfully completed.
- **Database Connection** – Used to connect the database that contains the tables needed to define and validate the cube.
- Provides support for **Type 2 Slowly-Changing Dimensions** by allowing multiple versions of the same record with validity intervals. -> Use **dimension lookup/update** step to configure the surrogate key, the version and the date range fields.
- **PROS**:
    - Offers a large set of operations.
    - Transformations are more explicit (than in SQL).
- **CON** – Join rows is faster in SQL.

**Data Cleaner**:

- Tool for data profiling.
- Can be used to inspect the data coming from the input data source.
- Can be used to detect anomalies and gather statistics about the data.
- Does not allow replacement of data.

**Pentaho Schema Workbench (PSW)**:

- Allows us to define a data cube together with its dimensions, hierarchies, levels, and measures, based on the relational schema of a data warehouse.
- **Database Connection** – Used to connect the database that contains the tables needed to define and validate the cube.
- The cube definition can be exported as an XML file that can be imported into Saiku.

**Saiku**:

- Web-based tool that allows us to perform MDX queries over a data cube, by means of a user-friendly drag-and-drop interface.
- OLAP tool.
- Used for exploratory analysis where the MDX query is being changed as the analysis proceeds.
- Means to gain insights into the business.
- A query (ad-hoc query) is performed by dragging (measures go to Measures and dimensions go to Rows or Columns).
- Generates the code for a MDX query and executes it over the data warehouse.
- It is a front-end for executing queries.

**Pentaho Report Designer (PRD)**:

- Used to automatically generate reports from a given data source (database / data warehouse).
- Reporting tool.
- Allows us to generate pre-defined reports from queries which are not supposed to change. If the MDX query changes, then it is probably a new report.
- Typically used to gather analytical data. => **Better to query the DW with MDX rather than with SQ**L because it facilitates the analysis of multi-dimensional data and because the DW contains pre-computed measures that can be used for such analysis (when querying with SQL, those measures have to be recomputed every time).
- **Database Connection** – Used to connect to the database where the data for the report comes from, using SQL or MDX queries.
- It is possible to build a single report with the results of a MDX and a SQL query.
- It is possible to define one or more data sources to be used in the report:
    - Each data source may have a different query.
    - The report can be configured to use the results of those queries for instance by drag-and-dropping its elements on the correct report sections.