Multidimensional Queries

Krzysztof Dembczyński

Intelligent Decision Support Systems Laboratory (IDSS)
Poznań University of Technology, Poland

Software Development Technologies
Master studies, first semester
Academic year 2016/17 (winter course)

Review of the Previous Lecture

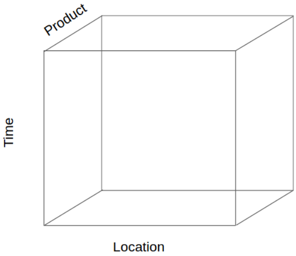
- Mining of massive datasets.
- Evolution of database systems.
- Dimensional modeling.
- ETL and OLAP systems:
  - Extraction, transformation, load.
  - ROLAP, MOLAP, HOLAP.
  - Challenges in OLAP systems: a huge number of possible aggregations to compute.

Motivation

- We need an intuitive way of expressing analytical (multidimensional) queries:

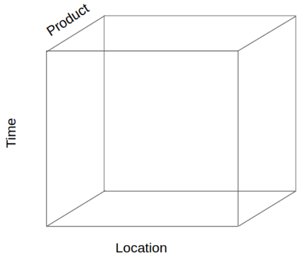
Motivation

- We need an intuitive way of expressing analytical (multidimensional) queries:
  - Operations like roll up, drill down, slice and dice, pivoting, ranking, time and window functions, etc.



Motivation

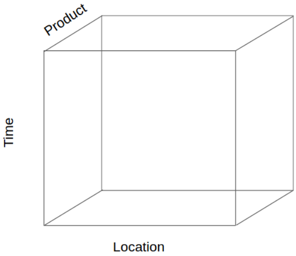
- We need an intuitive way of expressing analytical (multidimensional) queries:
  - Operations like roll up, drill down, slice and dice, pivoting, ranking, time and window functions, etc.



- Two solutions:

Motivation

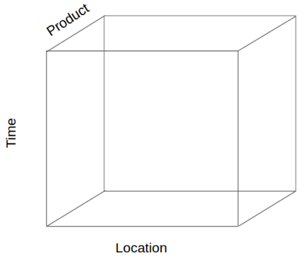
- We need an intuitive way of expressing analytical (multidimensional) queries:
    - Operations like roll up, drill down, slice and dice, pivoting, ranking, time and window functions, etc.



- Two solutions:
    - Extending **SQL**, or

Motivation

- We need an intuitive way of expressing analytical (multidimensional) queries:
  - Operations like roll up, drill down, slice and dice, pivoting, ranking, time and window functions, etc.



- Two solutions:
  - Extending **SQL**, or
  - Inventing a new language ($\rightarrow$ **MDX**).

Outline

Outline

OLAP Queries

- A typical example of an analytical query is a group-by query:

```
SELECT Instructor, Academic_year, AVG(Grade)
FROM Data_Warehouse
GROUP BY Instructor, Academic_year
```

- And the result:

Academic_year	Name	AVG(Grade)
2013/14	Stefanowski	4.2
2014/15	Stefanowski	4.5
2013/14	Słowiński	4.1
2014/15	Słowiński	4.3
2014/15	Dembczyński	4.6

SQL

- OLAP extensions in SQL:
  - GROUP BY CUBE,
  - GROUP BY ROLLUP,
  - GROUP BY GROUPING SETS,
  - GROUPING and DECODE/CASE
  - OVER and PARTITION BY,
  - RANK.

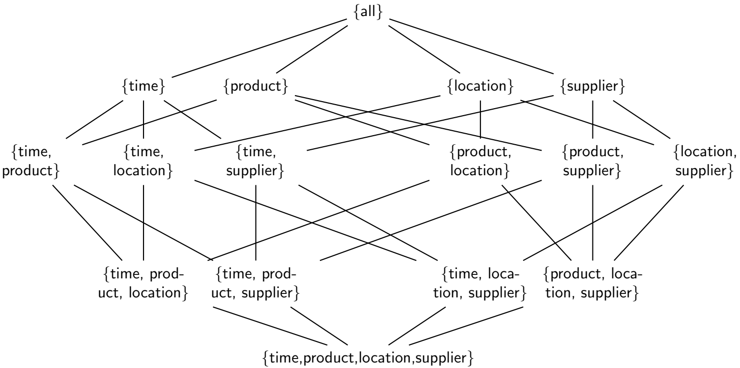
SQL

- `GROUP BY CUBE`

SQL

- `GROUP BY CUBE`
  - **Example**:

```
SELECT Time, Product, Location, Supplier, SUM(Gain)
FROM Sales
GROUP BY CUBE (Time, Product, Location, Supplier);
```



SQL

- `GROUP BY CUBE`
  - **Example**:

```
SELECT Time, Product, Location, Supplier, SUM(Gain)
FROM Sales
GROUP BY Time, Product, Location, Supplier
UNION ALL
SELECT Time, Product, Location, ''*'', SUM(Gain)
FROM Sales
GROUP BY Time, Product, Location
UNION ALL
SELECT Time, Product, ''*'', Location, SUM(Gain)
FROM Sales
GROUP BY Time, Product, Location
UNION ALL
...
UNION ALL
SELECT '*', '*', '*', '*', SUM(Gain)
FROM Sales;
```

SQL

- `GROUP BY CUBE`
  - It is not only a *Macro* instruction to reduce the number of subgroup-bys.

SQL

- `GROUP BY CUBE`
  - It is not only a *Macro* instruction to reduce the number of subgroup-bys.
  - One can easily optimize the group-by operations, when they are performed all-together: upper-level group-bys can be computed from lower-level group-bys.

SQL

- GROUP BY CUBE
  - **Example**:

```
SELECT Academic year, Name, AVG(Grade) FROM
Students_grades GROUP BY CUBE(Academic year, Name);
```

All rows and columns

Academic_year	Name	AVG(Grade)
2011/2	Stefanowski	4.2
2011/2	Słowiński	4.1
2012/3	Stefanowski	4.0
2012/3	Słowiński	3.8
2013/4	Stefanowski	3.9
2013/4	Słowiński	3.6
2013/4	Dembczyński	4.8

Academic_year	AVG(Grade)
2011/2	4.15
2012/3	3.85
2013/4	3.8

Name	AVG(Grade)
Stefanowski	3.9
Słowiński	3.6
Dembczyński	4.8

AVG(Grade)
3.95

SQL

- `GROUP BY CUBE`
  - **Example**:
    ```
    SELECT Academic year, Name, AVG(Grade) FROM
    Students_grades GROUP BY CUBE(Academic year, Name);
    ```

Academic_year	Name	AVG(Grade)
2011/2	Stefanowski	4.2
2011/2	Słowiński	4.1
2012/3	Stefanowski	4.0
2012/3	Słowiński	3.8
2013/4	Stefanowski	3.9
2013/4	Słowiński	3.6
2013/4	Dembczyński	4.8
2011/2	NULL	4.15
2012/3	NULL	3.85
2013/4	NULL	3.8
NULL	Stefanowski	3.9
NULL	Słowiński	3.6
NULL	Dembczyński	4.8
NULL	NULL	3.95

SQL

- `GROUP BY ROLLUP`

SQL

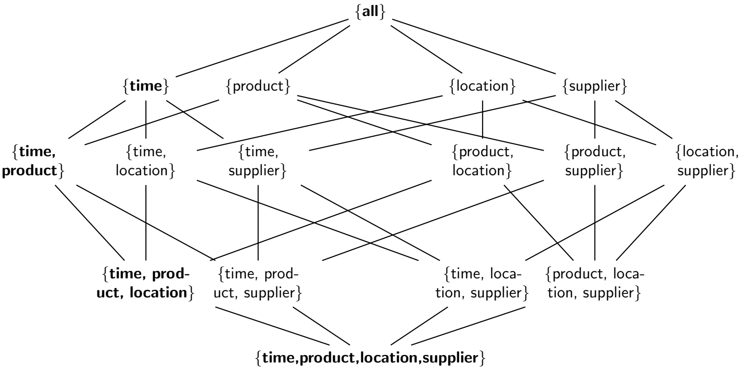
- GROUP BY ROLLUP
  - **Example**:

```
SELECT Time, Product, Location, Supplier, SUM(Gain)
FROM Sales
GROUP BY ROLLUP (Time, Product, Location, Supplier);
```

SQL

- GROUP BY ROLLUP
  - **Example**:

```
SELECT Time, Product, Location, Supplier, SUM(Gain)
FROM Sales
GROUP BY ROLLUP (Time, Product, Location, Supplier);
```



SQL

- GROUP BY ROLLUP
  - **Example**:

```
SELECT Time, Product, Location, Supplier, SUM(Gain)
FROM Sales
GROUP BY Time, Product, Location, Supplier
UNION ALL
SELECT Time, Product, Location, ''*'', SUM(Gain)
FROM Sales
GROUP BY Time, Product, Location
UNION ALL
SELECT Time, Product, ''*'', ''*'', SUM(Gain)
FROM Sales
GROUP BY Time, Product
UNION ALL
SELECT Time,''*'',''*'',''*'',SUM(Gain)
FROM Sales
GROUP BY Time
UNION ALL
SELECT '*', '*', '*', '*', SUM(Gain)
FROM Sales;
```

- GROUP BY ROLLUP
  - **Example**:
    ```
    SELECT Academic year, Name, AVG(Grade) FROM
    Students_grades GROUP BY ROLLUP(Academic year, Name);
    ```

Academic_year	Name	AVG(Grade)
2011/2	Stefanowski	4.2
2011/2	Słowiński	4.1
2012/3	Stefanowski	4.0
2012/3	Słowiński	3.8
2013/4	Stefanowski	3.9
2013/4	Słowiński	3.6
2013/4	Dembczyński	4.8
2011/2	NULL	4.15
2012/3	NULL	3.85
2013/4	NULL	3.8
NULL	NULL	3.95

SQL

- GROUP BY GROUPING SETS

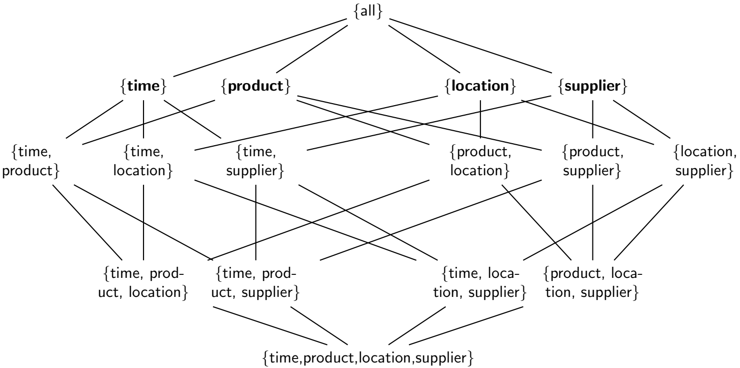
SQL

- GROUP BY GROUPING SETS
  - **Example**:
    ```
    SELECT Time, Product, Location, Supplier, SUM(Gain)
    FROM Sales
    GROUP BY GROUPING SETS (Time, Product, Location,
    Supplier);
    ```

SQL

- `GROUP BY GROUPING SETS`
  - **Example**:

```
SELECT Time, Product, Location, Supplier, SUM(Gain)
FROM Sales
GROUP BY GROUPING SETS (Time, Product, Location,
Supplier);
```



SQL

- `GROUP BY GROUPING SETS`

SQL

- GROUP BY GROUPING SETS
  - **Example**:

```
SELECT Time,''*'',''*'',''*'',SUM(Gain)
FROM Sales
GROUP BY Time
UNION ALL
SELECT ''*'',Product,''*'',''*'',SUM(Gain)
FROM Sales
GROUP BY Product
UNION ALL
SELECT ''*'',''*'', Location, ''*'',SUM(Gain)
FROM Sales
GROUP BY Location
UNION ALL
SELECT ''*'',''*'',''*'',Supplier, SUM(Gain)
FROM Sales
GROUP BY Supplier
```

SQL

- `GROUP BY GROUPING SETS`
  - **Example**:
    ```
    SELECT Academic year, Name, AVG(Grade) FROM
    Students_grades GROUPING SETS (Academic year, Name,());
    ```

Academic_year	Name	AVG(Grade)
2011/2	NULL	4.15
2012/3	NULL	3.85
2013/4	NULL	3.8
NULL	Stefanowski	3.9
NULL	Słowiński	3.6
NULL	Dembczyński	4.8
NULL	NULL	3.95

SQL

- The `NULL` returned as the result of a `ROLLUP`, `CUBE` or `GROUPING SETS` operation is a special use of `NULL` that represents *all values*.

SQL

- The `NULL` returned as the result of a `ROLLUP`, `CUBE` or `GROUPING SETS` operation is a special use of `NULL` that represents *all values*.
- How to distinguish this null value from a standard null?

SQL

- The `NULL` returned as the result of a `ROLLUP`, `CUBE` or `GROUPING SETS` operation is a special use of `NULL` that represents *all values*.
- How to distinguish this null value from a standard null?
- `GROUPING(<column_expression>)`

SQL

- The NULL returned as the result of a ROLLUP, CUBE or GROUPING SETS operation is a special use of NULL that represents *all values*.
- How to distinguish this null value from a standard null?
- GROUPING(<column_expression>)
  - Returns a value of 1 if the value of expression in the row is a null representing the set of all values.

SQL

- The NULL returned as the result of a ROLLUP, CUBE or GROUPING SETS operation is a special use of NULL that represents *all values*.
- How to distinguish this null value from a standard null?
- GROUPING(<column_expression>)
  - Returns a value of 1 if the value of expression in the row is a null representing the set of all values.
  - <column expression> is a column or an expression that contains a column in a GROUP BY clause.

SQL

- The NULL returned as the result of a ROLLUP, CUBE or GROUPING SETS operation is a special use of NULL that represents *all values*.
- How to distinguish this null value from a standard null?
- GROUPING(<column_expression>)
  - Returns a value of 1 if the value of expression in the row is a null representing the set of all values.
  - <column expression> is a column or an expression that contains a column in a GROUP BY clause.
- **Example**:

```
SELECT Scholarship, AVG(Grade), GROUPING(Scholarship) as
Grouping FROM Students grades GROUP BY ROLL UP(Scholarship);
```

Scholarship	AVG(Grade)	Grouping
Yes	4.15	0
No	3.61	0
NULL	4.03	0
NULL	3.89	1

SQL

- Use a `DECODE`-like function or `CASE`-like instruction to properly format your results.

SQL

- Use a `DECODE`-like function or `CASE`-like instruction to properly format your results.
- **Example**:

```
SELECT CASE
WHEN GROUPING(Scholarship) = 1 THEN "Total average"
WHEN GROUPING(Scholarship) = 0 THEN Scholarship
END AS Scholarship,
AVG(Grade),
FROM Grades
GROUP BY ROLL UP(Scholarship);
```

Scholarship	AVG(Grade)	Grouping
Yes	4.15	0
No	3.61	0
NULL	4.03	0
Total average	3.89	1

SQL

- `OVER()`:

SQL

- `OVER()`:
  - Determines the partitioning and ordering of a rowset before the associated window function is applied.

SQL

- `OVER()`:
  - Determines the partitioning and ordering of a rowset before the associated window function is applied.
  - The `OVER` clause defines a window or user-specified set of rows within a query result set.

SQL

- `OVER()`:
  - Determines the partitioning and ordering of a rowset before the associated window function is applied.
  - The `OVER` clause defines a window or user-specified set of rows within a query result set.
  - A window function then computes a value for each row in the window.

SQL

- `OVER()`:
  - Determines the partitioning and ordering of a rowset before the associated window function is applied.
  - The `OVER` clause defines a window or user-specified set of rows within a query result set.
  - A window function then computes a value for each row in the window.
  - The `OVER` clause can be used with functions to compute aggregated values such as moving averages, cumulative aggregates, running totals, or a top N per group results.

SQL

- `OVER()`:
  - Determines the partitioning and ordering of a rowset before the associated window function is applied.
  - The `OVER` clause defines a window or user-specified set of rows within a query result set.
  - A window function then computes a value for each row in the window.
  - The `OVER` clause can be used with functions to compute aggregated values such as moving averages, cumulative aggregates, running totals, or a top N per group results.
  - Syntax:

```
OVER (
    [ <PARTITION BY clause> ]
    [ <ORDER BY clause> ]
    [ <ROW or RANGE clause> ]
)
```

SQL

- `OVER()`:

SQL

- `OVER()`:
  - `PARTITION BY`:

SQL

- `OVER()`:
  - `PARTITION BY`:
    - Divides the query result set into partitions. The window function is applied to each partition separately and computation restarts for each partition.

SQL

- `OVER()`:
  - `PARTITION BY`:
    - Divides the query result set into partitions. The window function is applied to each partition separately and computation restarts for each partition.
  - `ORDER BY`:

SQL

- `OVER()`:
  - `PARTITION BY`:
    - Divides the query result set into partitions. The window function is applied to each partition separately and computation restarts for each partition.
  - `ORDER BY`:
    - Defines the logical order of the rows within each partition of the result set, i.e., it specifies the logical order in which the window function calculation is performed.

SQL

- `OVER()`:
  - `PARTITION BY`:
    - Divides the query result set into partitions. The window function is applied to each partition separately and computation restarts for each partition.
  - `ORDER BY`:
    - Defines the logical order of the rows within each partition of the result set, i.e., it specifies the logical order in which the window function calculation is performed.
  - `ROW` and `RANGE`:

SQL

- `OVER()`:
  - `PARTITION BY`:
    - Divides the query result set into partitions. The window function is applied to each partition separately and computation restarts for each partition.
  - `ORDER BY`:
    - Defines the logical order of the rows within each partition of the result set, i.e., it specifies the logical order in which the window function calculation is performed.
  - `ROW` and `RANGE`:
    - Further limits the rows within the partition by specifying start and end points within the partition.

SQL

- `OVER()`:
  - `PARTITION BY`:
    - Divides the query result set into partitions. The window function is applied to each partition separately and computation restarts for each partition.
  - `ORDER BY`:
    - Defines the logical order of the rows within each partition of the result set, i.e., it specifies the logical order in which the window function calculation is performed.
  - `ROW` and `RANGE`:
    - Further limits the rows within the partition by specifying start and end points within the partition.
    - This is done by specifying a range of rows with respect to the current row either by logical association or physical association.

SQL

- `OVER()`:
  - `PARTITION BY`:
    - Divides the query result set into partitions. The window function is applied to each partition separately and computation restarts for each partition.
  - `ORDER BY`:
    - Defines the logical order of the rows within each partition of the result set, i.e., it specifies the logical order in which the window function calculation is performed.
  - `ROW` and `RANGE`:
    - Further limits the rows within the partition by specifying start and end points within the partition.
    - This is done by specifying a range of rows with respect to the current row either by logical association or physical association.
    - The `ROWS` clause limits the rows within a partition by specifying a fixed number of rows preceding or following the current row.

SQL

- `OVER()`:
  - `PARTITION BY`:
    - Divides the query result set into partitions. The window function is applied to each partition separately and computation restarts for each partition.
  - `ORDER BY`:
    - Defines the logical order of the rows within each partition of the result set, i.e., it specifies the logical order in which the window function calculation is performed.
  - `ROW` and `RANGE`:
    - Further limits the rows within the partition by specifying start and end points within the partition.
    - This is done by specifying a range of rows with respect to the current row either by logical association or physical association.
    - The `ROWS` clause limits the rows within a partition by specifying a fixed number of rows preceding or following the current row.
    - The `RANGE` clause logically limits the rows within a partition by specifying a range of values with respect to the value in the current row.

SQL

- `OVER()`:
  - `PARTITION BY`:
    - Divides the query result set into partitions. The window function is applied to each partition separately and computation restarts for each partition.
  - `ORDER BY`:
    - Defines the logical order of the rows within each partition of the result set, i.e., it specifies the logical order in which the window function calculation is performed.
  - `ROW` and `RANGE`:
    - Further limits the rows within the partition by specifying start and end points within the partition.
    - This is done by specifying a range of rows with respect to the current row either by logical association or physical association.
    - The `ROWS` clause limits the rows within a partition by specifying a fixed number of rows preceding or following the current row.
    - The `RANGE` clause logically limits the rows within a partition by specifying a range of values with respect to the value in the current row.
    - Preceding and following rows are defined based on the ordering in the `ORDER BY` clause.

SQL

- **Examples**

SQL

- **Examples**
  - Moving average for a student:

```
SELECT Student, Academic_year,
AVG (grades) OVER (PARTITION BY Student ORDER BY
Academic_year DESC ROWS UNBOUNDED PRECEDING)
FROM Grades
ORDER BY Student, Academic_year;
```

- **Examples**
  - Moving average for a student:

```
SELECT Student, Academic_year,
AVG (grades) OVER (PARTITION BY Student ORDER BY
Academic_year DESC ROWS UNBOUNDED PRECEDING)
FROM Grades
ORDER BY Student, Academic_year;
```

  - Moving average for different departments:

```
SELECT Department, Academic_year,
AVG (grades) OVER (PARTITION BY Department ORDER BY
Academic_year DESC ROWS UNBOUNDED PRECEDING)
FROM Grades
ORDER BY Department, Academic_year;
```

SQL

- Ranking functions:

SQL

- Ranking functions:
  - `RANK () OVER`:

SQL

- Ranking functions:
  - `RANK () OVER`:
    - Returns the rank of each row within the partition of a result set. The rank of a row is one plus the number of ranks that come before the row in question.

SQL

- Ranking functions:
  - `RANK () OVER`:
    - Returns the rank of each row within the partition of a result set. The rank of a row is one plus the number of ranks that come before the row in question.
  - `DENSE RANK () OVER`:

SQL

- Ranking functions:
  - `RANK () OVER`:
    - Returns the rank of each row within the partition of a result set. The rank of a row is one plus the number of ranks that come before the row in question.
  - `DENSE RANK () OVER`:
    - Returns the rank of rows within the partition of a result set, without any gaps in the ranking. The rank of a row is one plus the number of distinct ranks that come before the row in question.

SQL

- Ranking functions:
  - `RANK () OVER`:
    - Returns the rank of each row within the partition of a result set. The rank of a row is one plus the number of ranks that come before the row in question.
  - `DENSE RANK () OVER`:
    - Returns the rank of rows within the partition of a result set, without any gaps in the ranking. The rank of a row is one plus the number of distinct ranks that come before the row in question.
  - `NTILE (integer_expression) OVER`:

SQL

- Ranking functions:
  - `RANK () OVER`:
    - Returns the rank of each row within the partition of a result set. The rank of a row is one plus the number of ranks that come before the row in question.
  - `DENSE RANK () OVER`:
    - Returns the rank of rows within the partition of a result set, without any gaps in the ranking. The rank of a row is one plus the number of distinct ranks that come before the row in question.
  - `NTILE (integer_expression) OVER`:
    - Distributes the rows in an ordered partition into a specified number of groups. The groups are numbered, starting at one. For each row, `NTILE` returns the number of the group to which the row belongs.

SQL

- Ranking functions:
  - `RANK () OVER`:
    - Returns the rank of each row within the partition of a result set. The rank of a row is one plus the number of ranks that come before the row in question.
  - `DENSE RANK () OVER`:
    - Returns the rank of rows within the partition of a result set, without any gaps in the ranking. The rank of a row is one plus the number of distinct ranks that come before the row in question.
  - `NTILE (integer_expression) OVER`:
    - Distributes the rows in an ordered partition into a specified number of groups. The groups are numbered, starting at one. For each row, `NTILE` returns the number of the group to which the row belongs.
  - `ROW NUMBER () OVER`:

SQL

- Ranking functions:
  - `RANK () OVER`:
    - Returns the rank of each row within the partition of a result set. The rank of a row is one plus the number of ranks that come before the row in question.
  - `DENSE RANK () OVER`:
    - Returns the rank of rows within the partition of a result set, without any gaps in the ranking. The rank of a row is one plus the number of distinct ranks that come before the row in question.
  - `NTILE (integer_expression) OVER`:
    - Distributes the rows in an ordered partition into a specified number of groups. The groups are numbered, starting at one. For each row, `NTILE` returns the number of the group to which the row belongs.
  - `ROW NUMBER () OVER`:
    - Returns the sequential number of a row within a partition of a result set, starting at 1 for the first row in each partition.

SQL

- **Examples**
  - Ranking of the students:

```
SELECT Student, Avg(Grade),
RANK () OVER (ORDER BY Avg(Grade) DESC)
FROM Grades GROUP BY Student;
```

- **Examples**
  - Ranking of the students:

```
SELECT Student, Avg(Grade),
RANK () OVER (ORDER BY Avg(Grade) DESC)
FROM Grades GROUP BY Student;
```

  - Ranking of students partitioned by instructors:

```
SELECT Instructor, Student, Avg(Grade),
RANK () OVER (PARTITION BY Instructor ORDER BY
Avg(Grade) DESC) AS ranks FROM Grades GROUP BY Student,
Instructor
ORDER BY Instructor, rank;
```

Outline

MDX

- MDX $\longrightarrow$ Multidimensional expressions.

MDX

- MDX $\longrightarrow$ Multidimensional expressions.
- For OLAP queries, MDX is an alternative to SQL:

MDX

- MDX $\longrightarrow$ Multidimensional expressions.
- For OLAP queries, MDX is an alternative to SQL:

MDX

- MDX $\longrightarrow$ Multidimensional expressions.
- For OLAP queries, MDX is an alternative to SQL:

Academic_year	Instructor	AVG(Grade)
2011/2	Stefanowski	4.2
2011/2	Słowiński	4.1
2012/3	Stefanowski	4.0
2012/3	Słowiński	3.8
2013/4	Stefanowski	3.9
2013/4	Słowiński	3.6
2013/4	Dembczyński	4.8

MDX

- MDX $\longrightarrow$ Multidimensional expressions.
- For OLAP queries, MDX is an alternative to SQL:

Academic_year	Instructor	AVG(Grade)
2011/2	Stefanowski	4.2
2011/2	Słowiński	4.1
2012/3	Stefanowski	4.0
2012/3	Słowiński	3.8
2013/4	Stefanowski	3.9
2013/4	Słowiński	3.6
2013/4	Dembczyński	4.8

$\downarrow$

AVG(Grade)	Academic_year		
Name	2011/2	2012/3	2013/4
Stefanowski	4.2	4.0	3.9
Słowiński	4.1	3.8	3.6
Dembczyński			4.8

MDX

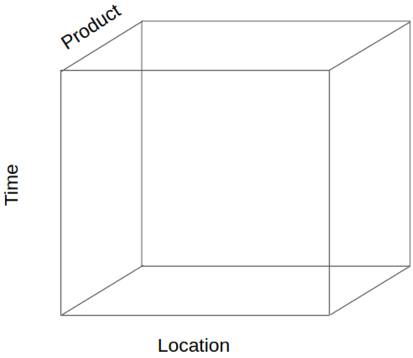
- MDX query:

```
SELECT {[Time].[1997],[Time].[1998]} ON COLUMNS,
{[Measures].[Sales],[Measures].[Cost]} ON ROWS
FROM Warehouse
WHERE ([Store].[All].[USA])
```

- Seems to be similar to **SQL**, but in fact it is quite **different**!

MDX

- Roll up – summarize data along a dimension hierarchy.
- Drill down – go from higher level summary to lower level summary or detailed data.
- Slice and dice – corresponds to selection and projection.
- Pivot – reorient cube.

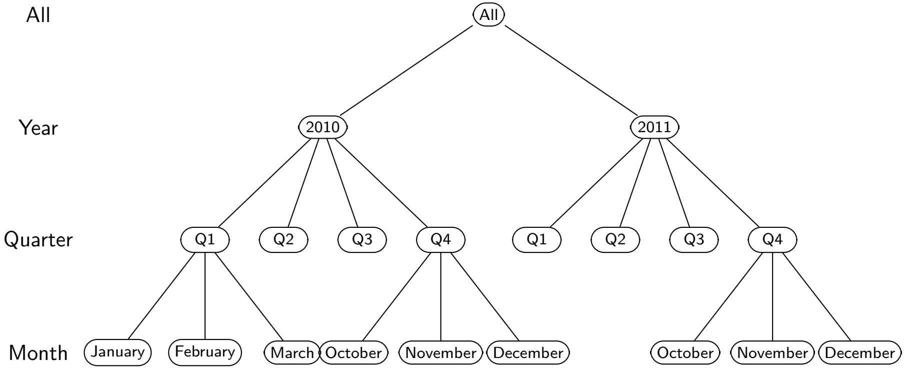


MDX

- Main concepts of MDX:
  - Dimension,
  - Measure,
  - Member,
  - Cell,
  - Hierarchy,
  - Aggregation,
  - Level,
  - Tuple,
  - Set,
  - Axis,
  - Member property.

Hierarchy

- **Example**: time hierarchy



MDX

- Identifying a member in a hierarchy:

  `[Time].[All].[2010].[Q1]`

  `[Store].[All].[Massachusetts].[Leominster]`

MDX

- Identifying a member in a hierarchy:

  `[Time].[All].[2010].[Q1]`

  `[Store].[All].[Massachusetts].[Leominster]`

- Short cuts possible:

  `[Store].[Leominster]`

MDX

- The concepts like dimension, measure, member, cell or hierarchy are intuitively well-understood.
- The concepts of tuple and set need more clarification.

MDX

- **Tuple**: An intersection of exactly a single member from each dimension (hierarchy) in the cube. For each dimension (hierarchy) that is not explicitly referenced, the *current member* is implicitly added to the tuple definition. A tuple always identifies a single cell in the multi-dimensional matrix. That could be an aggregate or a leaf level cell, but nevertheless one cell and only one cell is ever implied by a tuple.
- **Example**:

```
([Product].[Olives],[Store].[Poznan],[Time].[2014])
```

MDX

- **Set**: A set is a collection of tuples with the same dimensionality. It may have more than one tuple, but it can also have only one tuple, or even have zero tuples, in which case it is an empty set.
- **Example**:

```
{([Product].[Olives],[Store].[Poznan],[Time].[2013]),
([Product].[Olives],[Store].[Poznan],[Time].[2014]),
([Product].[Capers],[Store].[Poznan],[Time].[2013),
([Product].[Capers],[Store].[Poznan],[Time].[2014])}
```

MDX

- An MDX query must contain the following information:
  - The number of **axes** on which the result is presented.
  - The **set** of **tuples** to include on each axis of the MDX query.
  - The name of the **cube** that sets the context of the MDX query.
  - The set of members or tuples to include on the **slicer** axis.

MDX – Examples of queries

- **Query**:

```
SELECT
{[CARS].[All].[Chevy], [CARS].[All].[Ford]} ON ROWS,
{[DATE].[All].[March], [DATE].[All].[April]} ON COLUMNS
FROM Sales
WHERE [MEASURES].[SALES]
```

- **Result**:

	March	April
Chevy	$155 000.00	$ 75 000.00
Ford	$55 000.00	$175 000.00

MDX – Examples of queries

- **Query**:

```
SELECT
{[CARS].[All].[Chevy], [CARS].[All].[Ford]} ON ROWS,
{[DATE].[All].[March], [DATE].[All].[April]} ON COLUMNS
FROM Sales
WHERE [MEASURES].[SALES]
```

- **Result**:

	March	April
Chevy	$155 000.00	$ 75 000.00
Ford	$55 000.00	$175 000.00

Slices the cube by (Sales) with all not specified dimensions being set to default member

Defines a tuple (March) with all not specified dimensions being set to default member

Defines a tuple (Chevy) with all not specified dimensions being set to default member

Intersects all tuples to give (Chevy, March, Sales)

MDX – Examples of queries

- **Query**:

```
SELECT
{[CARS].[All].[Chevy], [CARS].[All].[Ford]} ON ROWS,
{[DATE].[All].[March], [DATE].[All].[April]} ON COLUMNS
FROM Sales
```

MDX – Examples of queries

- **Query**:

```
SELECT
{[CARS].[All].[Chevy], [CARS].[All].[Ford]} ON ROWS,
{[DATE].[All].[March], [DATE].[All].[April]} ON COLUMNS
FROM Sales
```

- **Result**:

	March	April
Chevy	$155 000.00	$ 75 000.00
Ford	$55 000.00	$175 000.00

MDX – Examples of queries

- **Query**:

```
SELECT
{[CARS].[All].[Chevy], [CARS].[All].[Ford]} ON ROWS,
{[DATE].[All].[March], [DATE].[All].[April]} ON COLUMNS
FROM Sales
WHERE ([MEASURES].[SALES_N])
```

MDX – Examples of queries

- **Query**:

```
SELECT
{[CARS].[All].[Chevy], [CARS].[All].[Ford]} ON ROWS,
{[DATE].[All].[March], [DATE].[All].[April]} ON COLUMNS
FROM Sales
WHERE ([MEASURES].[SALES_N])
```

- **Result**:

Sales_N	March	April
Chevy	1 000	700
Ford	600	1 500

- **Query**:

```
SELECT
{[CARS].[All].[Chevy], [CARS].[All].[Ford]} ON ROWS,
{DATE].[All].[JANUARY]:[DATE].[All].[APRIL]} ON COLUMNS
FROM Sales
```

MDX – Examples of queries

- **Query**:

```
SELECT
{[CARS].[All].[Chevy], [CARS].[All].[Ford]} ON ROWS,
{DATE].[All].[JANUARY]:[DATE].[All].[APRIL]} ON COLUMNS
FROM Sales
```

- **Result**:

	Chevy	Ford
January	$66 000.00	$ 79 000.00
February	$55 000.00	$72 000.00
March	$155 000.00	$55 000.00
April	$75 000.00	$175 000.00

MDX – Examples of queries

- **Query**:

```
SELECT
{[CARS].[All].[Chevy], [CARS].[All].[Ford]} ON ROWS,
{[DATE].[All].[YEAR].MEMBERS} ON COLUMNS
FROM Sales
```

MDX – Examples of queries

- **Query**:

```
SELECT
{[CARS].[All].[Chevy], [CARS].[All].[Ford]} ON ROWS,
{[DATE].[All].[YEAR].MEMBERS} ON COLUMNS
FROM Sales
```

- **Result**:

	Chevy	Ford
1998	$566 000.00	$479 000.00
1999	$545 000.00	$672 000.00
2000	$745 000.00	$ 527 000.00
2001	$345 000.00	$622 000.00

MDX – Examples of queries

- **Query**:

```
SELECT
{[CARS].[All].[Ford].CHILDREN} ON ROWS,
{[DATE].[All].[YEAR].MEMBERS]} ON COLUMNS
FROM Sales
```

MDX – Examples of queries

- **Query**:

```
SELECT
{[CARS].[All].[Ford].CHILDREN} ON ROWS,
{[DATE].[All].[YEAR].MEMBERS]} ON COLUMNS
FROM Sales
```

- **Result**:

	Ford Mustang	Ford Taurus	...
1998	$56 000.00	$79 000.00	
1999	$54 000.00	$72 000.00	
2000	$72 000.00	$52 000.00	
2001	$34 000.00	$22 000.00	

MDX – Examples of queries

- **Query**:

```
SELECT
{([CARS].[All].[CHEVY], [MEASURES].[SALES_SUM]),
([CARS].[All].[CHEVY], [MEASURES].[SALES_N]),
([CARS].[All].[FORD], [MEASURES].[SALES_SUM]),
([CARS].[All].[FORD], [MEASURES].[SALES_N]} ON ROWS,
{[DATE].[All].[YEAR].MEMBERS]} ON COLUMNS
FROM Sales
```

MDX – Examples of queries

- **Query**:

```
SELECT
{([CARS].[All].[CHEVY], [MEASURES].[SALES_SUM]),
([CARS].[All].[CHEVY], [MEASURES].[SALES_N]),
([CARS].[All].[FORD], [MEASURES].[SALES_SUM]),
([CARS].[All].[FORD], [MEASURES].[SALES_N]} ON ROWS,
{[DATE].[All].[YEAR].MEMBERS]} ON COLUMNS
FROM Sales
```

- **Result**:

	Chevy		Ford	
	Sales_Sum	Sales_N	Sales_Sum	Sales_N
1998	$566 000.00	450	$479 000.00	450
1999	$545 000.00	475	$672 000.00	670
2000	$745 000.00	750	$527 000.00	490
2001	$345 000.00	325	$622 000.00	640

MDX – Examples of queries

- **Query**:

```
SELECT
CROSSJOIN({[CARS].[ALL CARS].[CHEVY], [CARS].[ALL
CARS].[FORD]}, {[MEASURES].[SALES SUM], [MEASURES].[SALES N]}
) ON COLUMN, {[DATE].[All].[YEAR].MEMBERS]} ON COLUMNS
FROM Sales
```

MDX – Examples of queries

- **Query**:

```
SELECT
CROSSJOIN({[CARS].[ALL CARS].[CHEVY], [CARS].[ALL
CARS].[FORD]}, {[MEASURES].[SALES SUM], [MEASURES].[SALES N]}
) ON COLUMN, {[DATE].[All].[YEAR].MEMBERS]} ON COLUMNS
FROM Sales
```

- **Result**:

	Chevy		Ford	
	Sales_Sum	Sales_N	Sales_Sum	Sales_N
1998	$566 000.00	450	$479 000.00	450
1999	$545 000.00	475	$672 000.00	670
2000	$745 000.00	750	$527 000.00	490
2001	$345 000.00	325	$622 000.00	640

MDX – Examples of queries

- **Query**:

```
SELECT
NON EMPTY [Store Type].[Store Type].MEMBERS ON COLUMNS,
FILTER([Store].[Store City].MEMBERS, (Measures.[Profit],
[Time].[1997]) > 250000) ON ROWS
FROM [Sales]
WHERE (Measures.[Profit], [Time].[Year].[1997])
```

MDX – Examples of queries

- **Query**:

```
SELECT
NON EMPTY [Store Type].[Store Type].MEMBERS ON COLUMNS,
FILTER([Store].[Store City].MEMBERS, (Measures.[Profit],
[Time].[1997]) > 250000) ON ROWS
FROM [Sales]
WHERE (Measures.[Profit], [Time].[Year].[1997])
```

- **Result**:

Profit	Normal	24 hours
Toronto	$66 000.00	$196 000.00
Vancouver	$111 000.00	$156 000.00
New York	$59 000.00	$196 000.00
Chicago	$75 000.00	$ 211 000.00

MDX – Examples of queries

- **Query**:

```
SELECT
Measures.MEMBERS ON COLUMNS,
ORDER([Store].[Store City].MEMBERS, Measures.[Sales Count],
DESC) ON ROWS
FROM [Sales]
```

- **Query**:

```
SELECT
Measures.MEMBERS ON COLUMNS,
ORDER([Store].[Store City].MEMBERS, Measures.[Sales Count],
DESC) ON ROWS
FROM [Sales]
```

- **Result**:

	Profit	Sales Count
Toronto	$747 000.00	2 196 000
Vancouver	$785 000.00	1 956 000
New York	$666 000.00	1 916 000
Chicago	$711 000.00	1 596 000

MDX – Examples of queries

- **Query**:

```
WITH MEMBER
[Time].[Year Difference] AS [Time].[2nd half] - [Time].[1st half]
SELECT { [Account].[Income], [Account].[Expenses] } ON COLUMNS,
{ [Time].[1st half], [Time].[2nd half], [Time].[Year Difference] }
ON ROWS
FROM [Financials]
```

MDX – Examples of queries

- **Query**:

```
WITH MEMBER
[Time].[Year Difference] AS [Time].[2nd half] - [Time].[1st half]
SELECT { [Account].[Income], [Account].[Expenses] } ON COLUMNS,
{ [Time].[1st half], [Time].[2nd half], [Time].[Year Difference] }
ON ROWS
FROM [Financials]
```

- **Result**:

	Income	Expenses
1st Half	5 000	4 200
2nd Half	8 000	7 000
Year Difference	3 000	2 800

MDX – Examples of queries

- **Query**:

```
WITH MEMBER
[Account].[Net Income] AS
([Account].[Income] - [Account].[Expenses]) / [Account].[Income]
SELECT
[Account].[Income], [Account].[Expenses], [Account].[Net Income]
ON COLUMNS,
[Time].[1st half], [Time].[2nd half]  ON ROWS
FROM [Financials]
```

MDX – Examples of queries

- **Query**:

```
WITH MEMBER
[Account].[Net Income] AS
([Account].[Income] - [Account].[Expenses]) / [Account].[Income]
SELECT
[Account].[Income], [Account].[Expenses], [Account].[Net Income]
ON COLUMNS,
[Time].[1st half], [Time].[2nd half]  ON ROWS
FROM [Financials]
```

- **Result**:

	Income	Expenses	Net Income
1st Half	5 000	4 200	0.16
2nd Half	8 000	7 000	0.125

MDX – Examples of queries

- **Query**:

```
WITH MEMBER
[Time].[Year Difference] AS [Time].[2nd half] - [Time].[1st half],
SOLVE ORDER = 1
MEMBER [Account].[Net Income] AS
([Account].[Income] - [Account].[Expenses]) / [Account].[Income],
SOLVE ORDER = 2
SELECT
[Account].[Income], [Account].[Expenses], [Account].[Net Income]  ON
COLUMNS,
[Time].[1st half], [Time].[2nd half], [Time].[Year Difference]  ON
ROWS
FROM [Financials]
```

MDX – Examples of queries

- **Query**:

```
WITH MEMBER
[Time].[Year Difference] AS [Time].[2nd half] - [Time].[1st half],
SOLVE ORDER = 1
MEMBER [Account].[Net Income] AS
([Account].[Income] - [Account].[Expenses]) / [Account].[Income],
SOLVE ORDER = 2
SELECT
[Account].[Income], [Account].[Expenses], [Account].[Net Income]  ON
COLUMNS,
[Time].[1st half], [Time].[2nd half], [Time].[Year Difference]  ON
ROWS
FROM [Financials]
```

- **Result**:

	Income	Expenses	Net Income
1st Half	5 000	4 200	0.16
2nd Half	8 000	7 000	0.125
Year Difference	3 000	2 800	0.066

MDX – Examples of queries

- **Query**:

```
WITH MEMBER
[Time].[Year Difference] AS [Time].[2nd half] - [Time].[1st half],
SOLVE ORDER = 2
MEMBER [Account].[Net Income] AS
([Account].[Income] - [Account].[Expenses]) / [Account].[Income],
SOLVE ORDER = 1
SELECT
[Account].[Income], [Account].[Expenses], [Account].[Net Income]  ON
COLUMNS,
[Time].[1st half], [Time].[2nd half], [Time].[Year Difference]  ON
ROWS
FROM [Financials]
```

MDX – Examples of queries

- **Query**:

```
WITH MEMBER
[Time].[Year Difference] AS [Time].[2nd half] - [Time].[1st half],
SOLVE ORDER = 2
MEMBER [Account].[Net Income] AS
([Account].[Income] - [Account].[Expenses]) / [Account].[Income],
SOLVE ORDER = 1
SELECT
[Account].[Income], [Account].[Expenses], [Account].[Net Income]  ON
COLUMNS,
[Time].[1st half], [Time].[2nd half], [Time].[Year Difference]  ON
ROWS
FROM [Financials]
```

- **Result**:

	Income	Expenses	Net Income
1st Half	5 000	4 200	0.16
2nd Half	8 000	7 000	0.125
Year Difference	3 000	2 800	-0.035

MDX – Examples of queries

- **Query**:

```
WITH SET
[Quarter1] AS GENERATE([Time].[Year].MEMBERS, {
[Time].CURRENTMEMBER.FIRSTCHILD })
SELECT [Quarter1] ON COLUMNS,
[Store].[Store Name].MEMBERS ON ROWS
FROM [Sales]
WHERE (Measures.[Profit])
```

MDX – Examples of queries

- **Query**:

```
WITH SET
[Quarter1] AS GENERATE([Time].[Year].MEMBERS, {
[Time].CURRENTMEMBER.FIRSTCHILD })
SELECT [Quarter1] ON COLUMNS,
[Store].[Store Name].MEMBERS ON ROWS
FROM [Sales]
WHERE (Measures.[Profit])
```

- **Result**:

	2010Q1	2011Q2	...
Saturn	$ 147 000	$196 000	
Media Markt	$ 185 000	$156 000	
Avans	$ 166 000	$ 116 000	

SQL vs. MDX

- Single member:

SQL vs. MDX

- Single member:
  - SQL:

SQL vs. MDX

- Single member:
  - SQL: `where City = 'Redmond'`

SQL vs. MDX

- Single member:
  - SQL: `where City = 'Redmond'`
  - MDX:

SQL vs. MDX

- Single member:
  - SQL: `where City = 'Redmond'`
  - MDX: `[City].[Redmond]`

SQL vs. MDX

- Single member:
  - SQL: `where City = 'Redmond'`
  - MDX: `[City].[Redmond]`
- Multiple members (a set):

SQL vs. MDX

- Single member:
  - SQL: `where City = 'Redmond'`
  - MDX: `[City].[Redmond]`
- Multiple members (a set):
  - SQL:

SQL vs. MDX

- Single member:
  - SQL: `where City = 'Redmond'`
  - MDX: `[City].[Redmond]`
- Multiple members (a set):
  - SQL: `where City IN ('Redmond', 'Seattle')`

SQL vs. MDX

- Single member:
  - SQL: `where City = 'Redmond'`
  - MDX: `[City].[Redmond]`
- Multiple members (a set):
  - SQL: `where City IN ('Redmond', 'Seattle')`
  - MDX:

SQL vs. MDX

- Single member:
  - SQL: `where City = 'Redmond'`
  - MDX: `[City].[Redmond]`
- Multiple members (a set):
  - SQL: `where City IN ('Redmond', 'Seattle')`
  - MDX: { `([City].[Redmond]), ([City].[Seattle])` }

MDX

- SQL:

```
SELECT Sum(Sales), City FROM Sales
WHERE City IN ('Redmond', 'Seattle')
GROUP BY City
```

MDX

- SQL:

```
SELECT Sum(Sales), City FROM Sales
WHERE City IN ('Redmond', 'Seattle')
GROUP BY City
```

- MDX:

```
SELECT Measures.Sales ON 0,
NON EMPTY {([City].[Redmond]),([City].[Seattle])} ON 1
FROM Sales
```

MDX

- SQL:

```
SELECT Sum(Sales) FROM Sales
WHERE City IN ('Redmond', 'Seattle')
```

MDX

- SQL:

```
SELECT Sum(Sales) FROM Sales
WHERE City IN ('Redmond', 'Seattle')
```

- MDX:

```
SELECT Measures.Sales ON 0
FROM Sales
WHERE {([City].[Redmond]), ([City].[Seattle])}
```

Outline

Summary

- Two main approaches for querying data warehouses.
- ROLAP servers: SQL and its OLAP extensions.
- MOLAP servers: MDX.

Bibliography

- J. Han and M. Kamber. *Data Mining: Concepts and Techniques (second edition)*. Morgan Kaufmann Publishers, 2006
- Mark Whitehorn, Robert Zare, and Mosha Pasumansky. *Fast Track to MDX*. Springer, 2002