FUNDAMENTALS

1) Data Warehouse:

DW is a subject-oriented, integrated, non-volatile and time variant collection of data in support of management decisions.

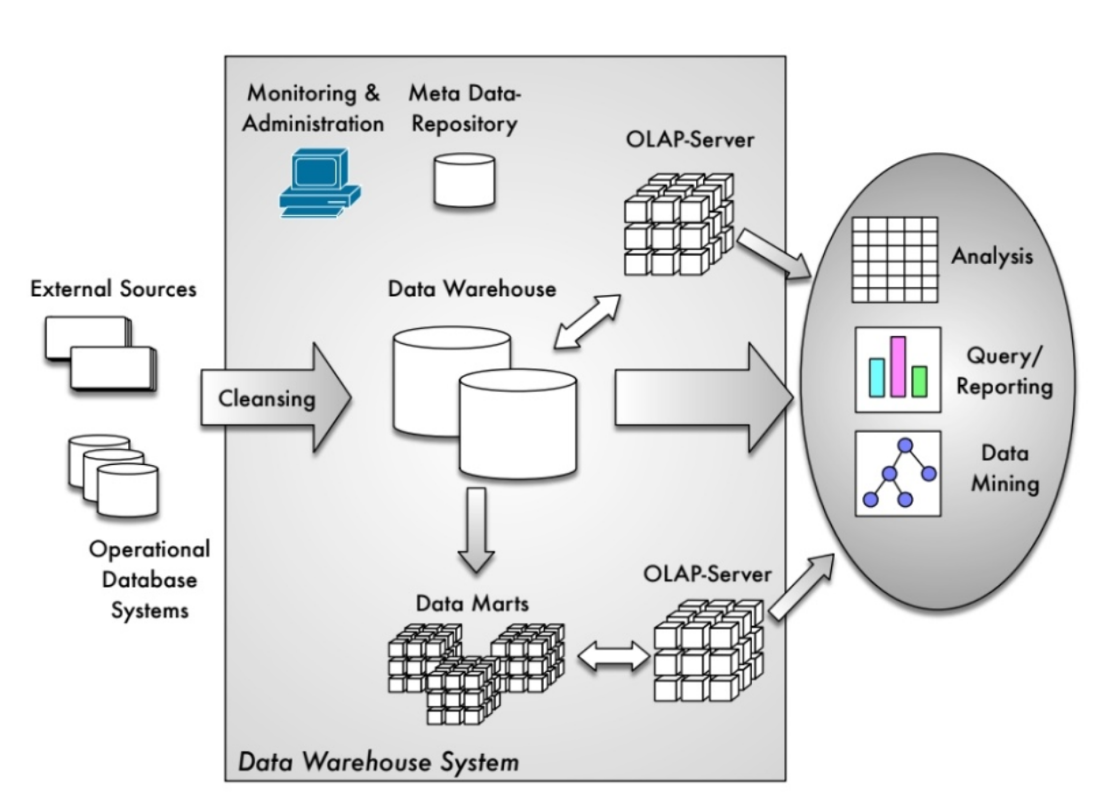
> A Data Warehouse is a **subject-oriented**, **integrated**, **non-volatile**, and **time variant** collection of data in support of managements decisions.
>
> *(W.H. Inmon 1996)*

In a narrower sense:
Core database and data cubes represent the data warehouse

> In a narrower sense:
> Core database and data cubes represent the data warehouse.

DW is a collection of data and technologies to support decision making

Data Warehouse: Collection of data and technologies to support decision making



Aspects of Data Warehouses

- Integration
  - Unification of data from different, mostly heterogeneous sources
  - Overcoming heterogeneity on different levels (system, schema, data)
- Analysis
  - Provision of data in a form desired by the user (related to decision area)
  - Requires preselection, time reference, aggregation

2) Data Warehousing: all steps involved for creation of a data warehouse (i.e) data retrieval (extraction), transformation, loading (or) saving, and analysis.

Data Warehousing

- Data Warehouse process, i.e., all steps of data retrieval (extraction, transformation, loading), saving and analysis

3) DWH Requirements:

Data Warehouse Requirements Include:

1) **Independence between data sources** and analytical systems (w.r.t. availability, load, ongoing changes)
2) **Integrated Data Storage** – Combines data from multiple sources into a consistent format. Continuous provision of integrated and derived data (Persistence)
3) **Reusability of provided data**
4) **Possibility to conduct arbitrary evaluations**
5) **Support of custom views (e.g.,w.r.t. time horizon,**

domain and structure)

6) **Extensibility (e.g., Integration of new sources)**
7) **Automation of processes**
8) **Uniqueness of data structures, access rights and processes**
9) **Orientation on the main purpose: data analysis**
10) **Time-Variant Data Storage** – Supports historical data analysis.
11) **Non-Volatile Data** – Data should not be frequently updated or deleted.
12) **Efficient Data Processing (ETL)** – Extracting, transforming, and loading data efficiently.
13) **High Scalability** – Ability to handle increasing data volumes.
14) **Security & Access Control** – Ensuring restricted and role-based access.
15) **Optimized for Analytical Queries (OLAP)** – Supports multidimensional views.

4) OLAP requirements (FASMI / Codd's rules)

Codd's 12 OLAP rules and the **FASMI criteria** define requirements for **Online Analytical Processing (OLAP)** systems, which are an essential part of **Data Warehousing**. However, they do not **fully** cover all **Data Warehouse requirements**—they primarily focus on the **analytical** aspects of data warehousing.

12 OLAP rules by Codd:

12 OLAP rules by Codd

- Multidimensional, conceptual view (1)
- Transparency (7)
- Accessibility (8)
- Performance (9)
- Scalability (10)
- Generic Dimensionality (2)
- Dynamic handling of sparse multidimensional structures (3)
- Multi-user mode (5)
- Unrestricted operations (11)
- Intuitive user interface (6)
- Flexible reporting (12)
- Any number of dimensionens und aggregation levels (4)

TAPS

Remember

1. **Multidimensional Conceptual View**: OLAP should allow users to analyze data from multiple perspectives (dimensions).
2. **Transparency**: The system should be easy to use, without requiring knowledge of underlying complexities.
3. **Accessibility**: Users should be able to retrieve data with minimal IT intervention.
4. **Consistent Reporting Performance**: Query performance should be consistent regardless of the number of dimensions or size of the dataset.
5. **Scalability**: The system should be scalable.
6. **Generic Dimensionality**: Each dimension should behave

similarly across different datasets.

7. **Dynamic Sparse Matrix Handling**: The system should efficiently handle large, sparse data cubes.
8. **Multi-User Support**: Multiple users should be able to access and analyze data simultaneously.
9. **Unrestricted Cross-Dimensional Operations**: Users should be able to perform calculations and comparisons across dimensions.
10. **Intuitive User Interface**: Users should be able to easily drill down, slice, dice, and pivot data.
11. **Flexible Reporting**: Users should be able to generate customized reports.
12. **Unlimited Dimensions & Aggregation Levels**: There should be no restrictions on the number of dimensions or levels of aggregation.

Relevance to Data Warehousing:
Codd's 12 OLAP rules focus on the analysis part of a Data Warehouse but do not address data integration, transformation, and loading (ETL), which are also critical parts of Data Warehousing.

FASMT :- Fast Analysis on Shared Multidimensional Information

FASMI

Fast Analysis on Shared Multidimensional Information

- Short response times (on average less than five seconds)
- Simple and flexible ways of evaluation
- Heterogeneous users with different rights
- Multidimensionality is an important criterion
- Questions on the number of required dimensions and ranges of values of associated attributes

- **Fast:** Queries should return results in less than 5 seconds.
- **Analysis:** Users should be able to perform complex calculations and statistical analyses.
- **Shared:** The system should support multi-user access with security and permissions.
- **Multidimensional:** The data model must support multiple dimensions (e.g., time, product, location).
- **Information:** The system should provide accurate and complete business data.

5) OLTP – Short Transaction (Online Transactional Processing)

OLAP – Long running Transaction (Online Analytical Processing)

OLAP (Online Analytical Processing)

- Explorative, interactive analysis based on the conceptual data model

OLTP	OLAP
1) Data Collection & Mgmt	1) Integration, consolidation & aggregation of multiple datasets
2) Short read/write access to few data records	2) Long running read transaction on many data sets
3) Responsibility of respective department	3) Analysis in the center

- Classical operational information systems
  → Online Transactional Processing (OLTP)
  - Data collection and management
  - Processing under responsibility of the respective department
  - Transactional processing: short read/write accesses to few data records
- Data Warehouse
  → Online Analytical Processing (OLAP)
  - Analysis in the center
  - Long-running read transactions on many data sets
  - Integration, consolidation and aggregation of data

	OLTP	OLAP
Focus	read, write, modify, delete	read, periodic insert
Transaction duration and type	short read/write transactions	long-lasting read transactions
Query structure	simple structured	complex
Volume of a query	few records	many records
Data model	query flexible	analysis-oriented
Data sources	usually one	more
Properties	non-derivative, current, autonomous, dynamic	derived / consolidated, historicized, integrated, stable
Data volume	MByte ... GByte	GByte ... TByte ... PByte
Accesses	single tuple access	table access (column by column)
User type	Input/Output by employee or application software	Manager, Controller, Analyst
User number	many	few (up to a few hundred)
Response time	msecs ... secs	secs ... min

6) Distinction DWH vs Big Data vs Parallel Systems vs Federated Systems

Definition: DBMS Techniques

- Parallel Databases
  - Technique for the implementation of a DWH
- Distributed databases
  - Usually no redundant data management
  - Distribution as a means of load distribution
  - No content integration/consolidation of data
- Federated Databases
  - Greater autonomy and heterogeneity
  - No specific analytical purpose
  - No read access optimization

Comparison: Data Warehouse (DWH) vs. Big Data vs. Parallel Systems vs. Federated Systems

These four systems are designed for **storing, processing, and analyzing large-scale data**, but they differ in **architecture, data processing methods, scalability, and integration capabilities**.

---

1. Data Warehouse (DWH)

Definition:

A **Data Warehouse (DWH)** is a **centralized repository** designed for **storing structured, historical data** for analysis and reporting. It integrates data from multiple sources and enables **fast analytical queries** using OLAP (Online Analytical Processing).

Key Features:

- **Structured Data:** Stores highly structured, cleaned, and transformed data.
- **ETL (Extract, Transform, Load):** Data is preprocessed before loading.
- **OLAP Support:** Optimized for complex **analytical queries** (aggregations, reports).
- **Non-Volatile & Time-Variant:** Stores historical data for trend analysis.
- **Example Systems:** Amazon Redshift, Google BigQuery, Snowflake, Oracle Data Warehouse.

---

2. Big Data Systems

Definition:

Big Data systems handle **large-scale, diverse, and fast-growing data** that may be **structured, semi-structured, or unstructured**. They use **distributed computing** for efficient storage and processing.

Key Features:

- **Handles 3Vs of Big Data:**
  - **Volume:** Terabytes to petabytes of data.
  - **Velocity:** High-speed data generation (real-time streaming).
  - **Variety:** Different data formats (text, images, videos, logs).
- **Schema-on-Read:** Data is stored **raw** and structured when queried.

- **Parallel Processing:** Uses **MapReduce, Spark, Hadoop** for distributed processing.
- **NoSQL & Distributed Storage:** MongoDB, HDFS, Apache Cassandra for scalability.
- **Example Systems:** Apache Hadoop, Apache Spark, Google Bigtable, Amazon S3.

---

3. Parallel Systems

Definition:

A **Parallel Database System** improves performance by distributing **data storage and processing tasks** across **multiple processors or servers**. It is often used in **DWH and Big Data** systems.

Key Features:

- **Parallel Query Execution:** Multiple CPUs work together to process queries faster.
- **Types of Parallelism:**
  - **Shared-Memory:** Multiple processors access a single memory.
  - **Shared-Disk:** Multiple nodes share a common storage system.
  - **Shared-Nothing:** Nodes have independent memory and storage (e.g., Google BigQuery).
- **Fault Tolerance:** If one processor fails, others continue working.
- **Example Systems:** Oracle Parallel Server, Greenplum, Microsoft SQL Server PDW.

---

4. Federated Systems

Definition:

A **Federated Database System (FDBS)** is an **integrated system of multiple independent databases** that function as a **single virtual database** while remaining autonomous.

Key Features:

- **Heterogeneous Databases:** Can integrate **SQL, NoSQL, and legacy systems**.
- **No Central Data Warehouse:** Queries are executed across multiple sources dynamically.
- **Uses Middleware:** A **federated query engine** processes data from different databases.
- **Example Systems:** IBM InfoSphere Federation Server, Apache Drill, Microsoft PolyBase.

Comparison Table: DWH vs. Big Data vs. Parallel vs. Federated Systems

Feature	Data Warehouse (DWH)	Big Data Systems	Parallel Systems	Federated Systems
Data Type	Structured	Structured, Semi-structured, Unstructured	Structured	Structured, Semi-structured
Storage Model	Centralized	Distributed	Centralized or Distributed	Decentralized
Processing	Batch & OLAP Queries	Real-time, Batch, Stream Processing	Parallel Query Execution	Distributed Queries
Schema	Schema-on-Write (Predefined)	Schema-on-Read (Flexible)	Schema-on-Write	Varies (Heterogeneous Databases)
Scalability	Limited (Scale-Up)	High (Scale-Out)	High (Scale-Out)	High (Integrates Multiple Databases)
Use Case	Business Intelligence, Reporting	IoT, Social Media, Log Processing	High-Speed Query Processing	Multi-System Data Integration
Example Systems	Amazon Redshift, Snowflake	Hadoop, Spark, BigQuery	Oracle Parallel Server	IBM InfoSphere, Apache Drill

---

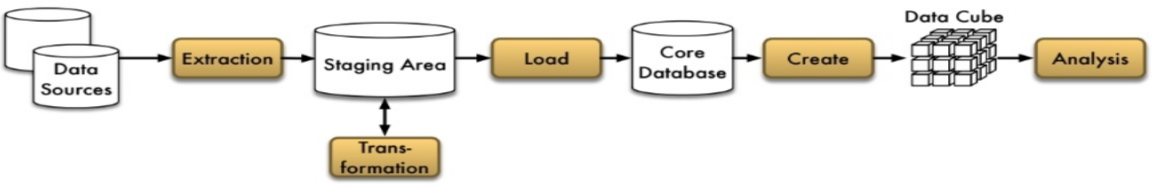
Final Thoughts: Which One to Use?

- ✔ **Use a Data Warehouse (DWH)** if you need **structured, historical data** for **BI and OLAP queries**.
- ✔ **Use Big Data Systems** if you deal with **large-scale, fast-changing, and unstructured data**.
- ✔ **Use Parallel Systems** when performance is key, and you need **fast, large-scale analytics**.
- ✔ **Use Federated Systems** if you need to **integrate multiple independent databases** dynamically.

ARCHITECTURE

DW Architecture

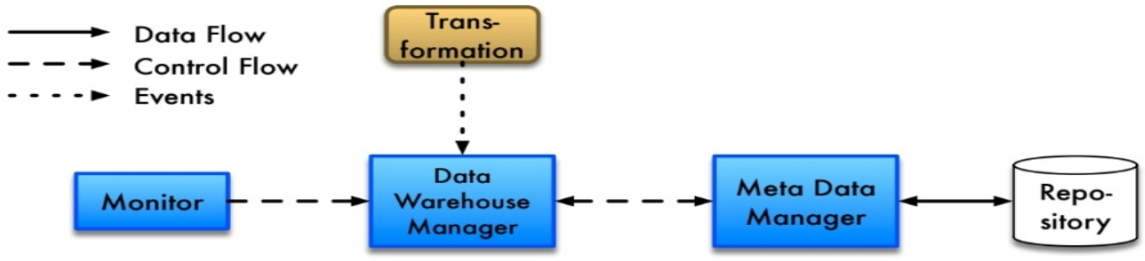
- Components of DW and their tasks
- Databases
  - Data sources: Origin of the data
  - Staging Area: temporary database for transformation
  - Data Warehouse: physical database for analysis
  - Repository: Database with metadata



+

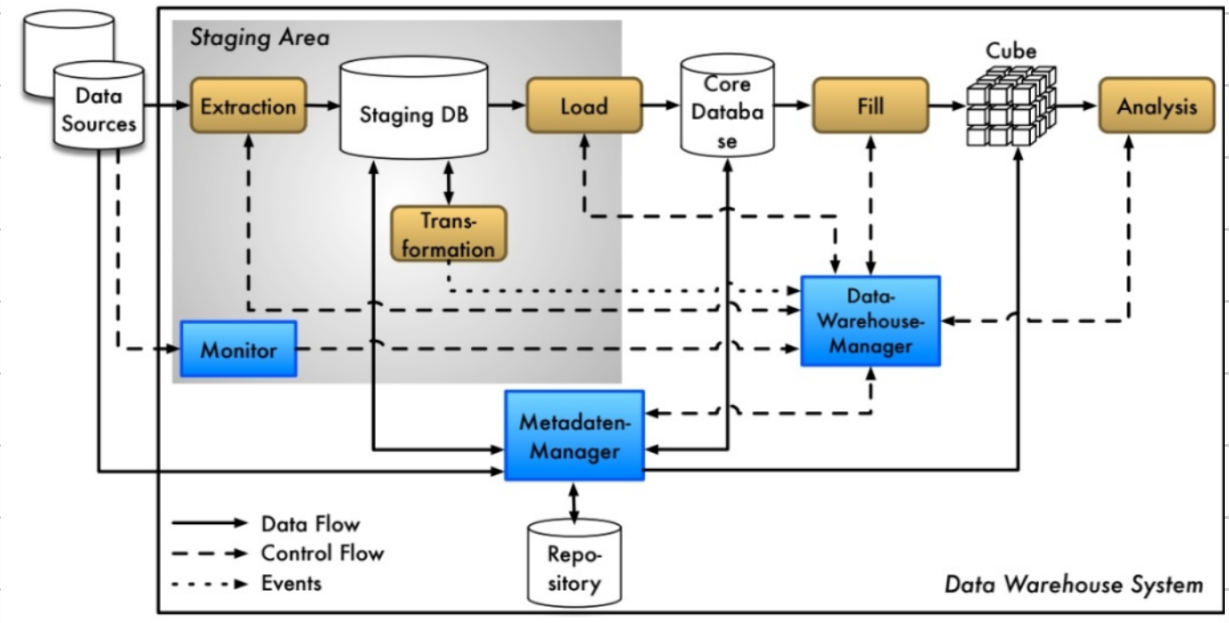
DW Architecture: Components

- Data-Warehouse-Manager: central control and management
- Monitors: Monitoring sources for changes
- Extractors: Selection and transport of data from sources in Data cleansing area
- Transformers: Standardization and data cleansing
- Loading components: Loading the transformed data into the DW
- Analysis components: Analysis and presentation of data



Learn definitions & characteristics

Full Architecture:



2) Components: Learn definition & characteristics

1) Data Warehouse Manager
2) Data Sources
3) Monitors
4) Staging DB
5) Extraction
6) Transformation
7) Loading
8) Core Database
9) Data Cube
10) Data Warehouse
11) Data Marts (X not in Architecture)
12) Analysis Tools
13) Repository
14) Metadata Manager

MULTIDIMENSIONAL DATA MODEL

1) Cube:

1. **Describe the data cube**, how is it defined and which operations does it support?

A **data cube** (or cuboid) in data warehousing is a foundational structure for multidimensional analysis. It represents data across multiple dimensions (e.g., product, location, time) as edges, with each cell in the cube containing one or more metrics related to those dimensions (e.g., revenue, profit).

- A **data cube** (or cuboid) is a multidimensional array of values (facts) that is defined by several dimensions.
- Formally, if you have a set of dimensions D1,D2,...,Dn and a set of metrics M, the cube is represented as:

  C=({D1,D2,...,Dn},{M1,M2,...,Mm})

  where each cell in the cube represents the measure(s) for a specific combination of dimension values.

Key Points:

- **Dimensions:** These are perspectives or axes (e.g., Time, Product, Geography).
- **Facts/Metrics:** The numerical measures such as revenue, sales quantity, or cost.
- **Visualization:** With two dimensions, it's a table; three

dimensions can be visualized as a dice; more than three, we refer to them as multidimensional cubes.

2. Supported operations include:

- **Pivoting (Rotation)** – Rotates the axes of the cube to view data from different perspectives. Swaps dimensions to analyze data from various perspectives.
- **Roll-Up** – Aggregates data by climbing up a hierarchy or dimension. For example, rolling up from daily to monthly data. Aggregates data along a hierarchy (e.g., day to month).
- **Drill-Down** – The opposite of roll-up, this operation increases the level of detail by navigating down the hierarchy (e.g., from quarterly to monthly sales). Moves from summarized to detailed data.
- **Drill-Across** – Allows switching between related cubes.
- **Slice** – Extracts a sub-cube by fixing one dimension's value (e.g., analyzing data for a single product). Extracts a subset of the cube by conditioning a dimension. WHERE dimA has a specific value
- **Dice** – Selects a smaller cube subset by applying conditions across multiple dimensions - WHERE dimA dimB and dimC has a specific range

3. Assign the following facts to the corresponding summation type, and explain the summation type.

- **Order quantity of an article per day** – **Flow**: Measures change over time, allowing aggregation across any period.

- **Stock** – Stock: Represents a status at a given time; aggregation is allowed except over time.
- **Exchange rate** – Value per Unit (VPU): A point-in-time measure, not cumulatively aggregable, with allowed operations being MIN, MAX, and AVG

Detailed explanation

FLOW:

- **Definition:** Measures that represent quantities over a period of time (flow data) that are aggregatable over time.
- **Allowed Aggregations:** SUM, COUNT, MIN, MAX, AVG.
- **Example:** Daily sales orders or production counts—summing these over days gives you the total orders over a period.

STOCK:

- **Definition:** Measures that represent a state at a specific point in time (stock data). Although you can sum these values over non-temporal dimensions, care must be taken when aggregating across time.
- **Allowed Aggregations:** SUM, COUNT, MIN, MAX, AVG (with caution—usually the snapshot at a point in time is taken).
- **Example:** Inventory levels (you usually consider the value at the end of the day, not summed over time).

VALUE PER UNIT (VPU):

- **Definition:** Measures that are not additive across any dimension because they represent a ratio or rate (point-

in-time measurements).

- **Allowed Aggregations:** Typically only MIN, MAX, or AVG are meaningful.
- **Example:** Price per item or exchange rate. Adding prices together does not make sense, but you can compute the average price.

4. Classify and define the following aggregate functions:

Distributive Aggregate Functions:

- **Definition:** Functions where the result of aggregating the entire dataset can be derived from aggregating subsets of the data.
- **Examples:** SUM, COUNT, MIN, MAX.
- **Property:** They can be computed in parts and then combined.
- **Example:** If you have sales for two regions, the overall sales is simply the sum of the two regional sums.

Algebraic Aggregate Functions: => derived

- **Definition:** Functions that can be computed from a fixed number of intermediate values.
- **Examples:** AVG (which can be computed as SUM/COUNT), STDDEV.
- **Property:** They require a combination of distributive functions. For instance, to calculate AVG, you first compute SUM and COUNT.

Holistic Aggregate Functions:

- **Definition:** Functions where no fixed-size summary of the data can be used to compute the result; the entire dataset might be needed.
- **Examples:** MEDIAN, RANK, PERCENTILE.
- **Property:** They are not decomposable; you can't simply combine pre-aggregated parts to get the final result.
- **Example:** To compute the median, you generally need to look at all individual values.

- Min – Distributive: Calculates the minimum value across data.
- Sum – Distributive: Adds values, fully aggregable.
- STDDEV – Algebraic: Measures data spread, requiring initial values to compute.
- Percentile – Holistic: Identifies data ranking, needing all data points.
- Count – Distributive: Counts entries, aggregable by partial sums.
- Rank – Holistic: Orders data, not distributive.
- Max – Distributive: Determines maximum value, aggregable.
- Average – Algebraic: Calculates mean, requiring sum and count.
- Median – Holistic: Middle value in ordered data, needing entire dataset

5. Describe different forms of hierarchies w.r.t. dimensions in a DWH.

There are two main forms of hierarchies:

- **Simple Hierarchies** – Present a straightforward classification with each level aggregating directly into the next. For instance, “Product Category → Product Family → Product Group → Item” shows successive levels leading up to a top node.
- **Parallel Hierarchies** – Within a single dimension, multiple, independent paths coexist without hierarchical dependence, such as a time dimension that includes separate paths for “Year → Quarter → Month → Day” and “Week,” allowing flexible aggregation levels - can use roll up or drill down to navigate in between the levels

6. Define the terms star-schema and snowflake-schema. What are their benefits and drawbacks?

Star-Schema: A denormalized schema where each dimension has a single table, optimizing query speed but increasing redundancy.

- Benefits: Simplified query structure, efficient within-dimension queries.
- Drawbacks: Redundant data, potential for anomalies in updates.

Snowflake-Schema: A normalized schema, with multiple tables for dimensions, reducing redundancy but requiring more joins.

- Benefits: Reduced data redundancy, efficient for large dimension tables.
- Drawbacks: More complex queries due to required joins, potentially

7. Slowly Changing Dimensions (SCDs)

Based on your lecture slides, here's a detailed explanation of **Slowly Changing Dimensions (SCDs) with a change in hierarchy level**:

---

1. What is a Slowly Changing Dimension (SCD)?

- A **Slowly Changing Dimension (SCD)** is a dimension that changes over time but not frequently.
- These changes can affect **attributes** (like a customer's address) or **hierarchies** (like an employee moving to a different department).
- Managing SCDs is crucial for maintaining historical data integrity in a data warehouse.

---

2. Types of Slowly Changing Dimensions (SCDs)

According to your lecture materials, SCDs can be classified into several types:

SCD Type	Description	Use Case
Type 1	Overwrites old data with new data	When history is not needed (e.g., fixing a typo)
Type 2	Creates a new row with a new surrogate key to maintain history	When tracking historical changes is essential (e.g., employee promotions)
Type 3	Adds a new column to store the previous value	Used when only a limited history is required (e.g., storing current and previous addresses)

---

3. Change in Hierarchy Level in SCDs

A **change in hierarchy level** means that a **dimension moves within its hierarchy**, affecting how data is grouped and aggregated.

Example Scenario: Employee Dimension with Department Hierarchy

- Imagine an **Employee Dimension** with a hierarchy:
  - **Company → Department → Team → Employee**

- If an employee moves from one department to another, this changes the **hierarchical relationship**.

---

4. How to Handle a Change in Hierarchy in an SCD?

There are **three main approaches** depending on how historical data should be stored:

✔ SCD Type 1: Overwrite the Old Relationship

- **What happens?** The old department is replaced with the new one.
- **Impact:** No historical record is kept.
- **Example:**
  - Before: Employee A → Sales Department
  - After: Employee A → Marketing Department
  - The data warehouse updates the record to reflect only the latest department.
- **Best Use Case:** When historical department tracking is not required.

✔ SCD Type 2: Create a New Record with a New Surrogate Key

- **What happens?** A new row is added in the dimension table to track the change.
- **Impact:** Historical reports will show employees in the department they belonged to at that time.
- **Example:**
  - Before:

```yaml
Emp_ID  | Name    | Department | Valid_From | Valid_To
-----------------------------------------------------
1001    | Alice   | Sales      | 2021-01-01 | 2023-06-30
```

  - After:

```yaml
Emp_ID  | Name    | Department | Valid_From | Valid_To
-----------------------------------------------------
1001    | Alice   | Sales      | 2021-01-01 | 2023-06-30
1002    | Alice   | Marketing  | 2023-07-01 | NULL
```

- **Best Use Case:** When tracking history is important (e.g., HR analytics, promotions, role changes).

✔ SCD Type 3: Store a Previous Hierarchy Level in a Separate Column

- **What happens?** A new column, `Previous_Department`, is added.
- **Impact:** Limited history is kept within a single row.
- **Example:**

```markdown
Emp_ID  | Name  | Department  | Previous_Department
-----------------------------------------------
1001    | Alice | Marketing   | Sales
```

- **Best Use Case:** When **only the last change** needs to be tracked.

---

5. Handling Hierarchical Changes in Fact Tables

- Since Fact tables store **foreign keys** from dimension tables, a hierarchy change might impact **aggregations** in OLAP reports.
- **With Type 1:** Aggregations change instantly as historical relationships are lost.
- **With Type 2:** The fact table still points to the previous dimension key, preserving historical aggregations.
- **With Type 3:** Aggregations might require extra logic to decide whether to use the current or previous hierarchy.

---

6. Summary

Approach	How it Handles Change in Hierarchy	Best Use Case
SCD Type 1	Overwrites the department, no history	Minor changes where history is not needed
SCD Type 2	Inserts a new row with a new surrogate key	Full historical tracking (e.g., Employee role changes)
SCD Type 3	Adds a new column to track the previous value	When limited history is sufficient

---

Exam Tip 📝

- If asked about **hierarchy changes in SCD**, always mention **SCD Type 2** as the **preferred method** for maintaining historical integrity.
- **Real-world example:** If a product moves from **Electronics** → **Home Appliances**, SCD Type 2 ensures past sales are still reported under "Electronics" while new sales fall under "Home Appliances."

Slowly changing Dimensions:

Analysis Requirements:

1) as is → current structure, easy, no history, memory <

memory> • as is & quasi 'as of' scenario → one additional column to store history without time allocation

memory>> • as is & 'as of' scenario: Versioning → Additional dim key required & history without time allocation

Drink_ID	Product	Product Group	Version	D_ID_old	active
1	Dornfelder	Red Wine	1	1	X
2	Müller-Thurgau	White Wine	1	2	X
3	Portugieser Weissherbst	Red Wine	1	3	X

Drink_ID	Product	Product Group	Version	D_ID_old	aktive
1	Dornfelder	Red Wine	1	1	X
2	Müller-Thurgau	White Wine	1	2	X
3	Portugieser Weissherbst	Red Wine	1	3	
3	Portugieser Weissherbst	White Wine	2	6	X

→ old row still exists but marked inactive

new version is introduced

memory>>> • as is & 'as of' scenario: Timestamps → Additional dim key req'd & history with time allocation

G_ID	D_ID_old	Product	Product Group	Valid_From	Valid_Until
1	1	Dornfelder	Red Wine	01.01.2019	31.12.9999
2	2	Müller-Thurgau	White Wine	01.01.2019	31.12.9999
3	3	Portugieser Weissherbst	Red Wine	01.01.2019	31.12.9999

G_ID	D_ID_old	Product	Product Group	Valid_From	Valid_Until
1	1	Dornfelder	Red Wine	01.01.2019	31.12.9999
2	2	Müller-Thurgau	White Wine	01.01.2019	31.12.9999
3	3	Portugieser Weissherbst	Red Wine	01.01.2019 →	31.12.2019
3	6	Portugieser Weissherbst	White Wine	01.01.2020 →	31.12.9999

2) As posted → memory>>> (historical truth)

Additional dim key & history of transaction date

Drink_ID	Product	Product Group	Version	D_ID_old
1	Dornfelder	Red Wine	1	1
2	Müller-Thurgau	White Wine	1	2
3	Portugieser Weissherbst	Red Wine	1	3

Drink_ID	Product	Product Group	Version	D_ID_old
1	Dornfelder	Red Wine	1	1
2	Müller-Thurgau	White Wine	1	2
3	Portugieser Weissherbst	Red Wine	1	3
3	Portugieser Weissherbst	White Wine	2	6

3) As is & as of & as posted: Snapshot

memory >>> in both dim & fact tables

Flag for currently active or not

Additional data for load time

recommended only for small dimensi

Drink_ID	Product	Product Group	Load Date	Current
1	Dornfelder	Red Wine	01.01.2019	0
2	Müller-Thurgau	White Wine	01.01.2019	0
3	Portugieser Weissherbst	Red Wine	01.01.2019	0
1	Dornfelder	Red Wine	01.01.2020	1
2	Müller-Thurgau	White Wine	01.01.2020	1
3	Portugieser Weissherbst	White Wine	01.01.2020	1

ETL

1) ETL:

- Extraction: selecting a section of the data from the sources and providing it for transformation
- Transformation: fitting the data to predefined schema and quality requirements
- Load: physical insertion of the data from the staging area into the data warehouse (including necessary aggregations)

ETL Process

- Frequently most elaborate part of the Data Warehousing
  - Variety of sources
  - Heterogeneity
  - Data volume
  - Complexity of the transformation
    - Schema and instance integration
    - Data cleansing
  - Hardly consistent methods and system support, but variety of tools available

2) Monitoring & Extraction techniques for different source types.

Extraction :-

1) Regular extraction of change data from sources
2) Data provision for DWH
3) Distinction of time of extraction and type of extracted data

Techniques overall → Point in Time

- **Synchronous** notification
  - Source propagates each change → on its own synchronously
- **Asynchronous** notification
  - Periodically
    - ★ Sources produce extracts regularly on defined period
    - ★ DWH regularly scans dataset
  - Event-driven
    - ★ DWH requests changes before each annual reporting } trigger based
    - ★ Source informs after each X changes
  - Query-controlled
    - ★ DWH queries for changes before any actual access

Different Source types of data

1) Flow → integrate all changes in DWH

2) Stock → point in time is essential & must be set

3) Value per Unit → Depending on unit & other dimensions

4) Snapshots → Source always provides complete data set (DSP).

5) Logs → Source provides any change.

Source ...		Method	Timeliness DWH	Workload on DWH	Worload on Sources
creates files periodically		Batch runs, Snapshots	depending on frequency	low	low
propagates each change		Trigger, Replication	maximum	high	very high
creates extracts on request	before use	very hard	maximum	medium	medium
	application-driven	application-driven	depending on frequency	depending on frequency	depending on frequency

Stock

flow

PIT event driven or query controlled

Extraction from Legacy Systems

- Dependent on the application
- Access to host system without online access
- Data in non-standard databases without APIs
- Unclear semantics → lack of domain knowledge

Solution → Commercial (or) custom tools to be used for extraction.

Bulk Loading

→ no integrity constraints

Definition: A method of loading large volumes of data into a database or Data Warehouse in a single operation rather than row by row. DB-specific extensions for loading large amounts of data

Characteristics:

- Typically used for **initial data warehouse loads** or

periodic **batch updates**.

- Loads **millions or billions of records** at once.
- Bypasses logging and indexing processes to improve speed.
- Uses optimized tools like SQL*Loader (Oracle), bcp (SQL Server), or COPY (PostgreSQL).

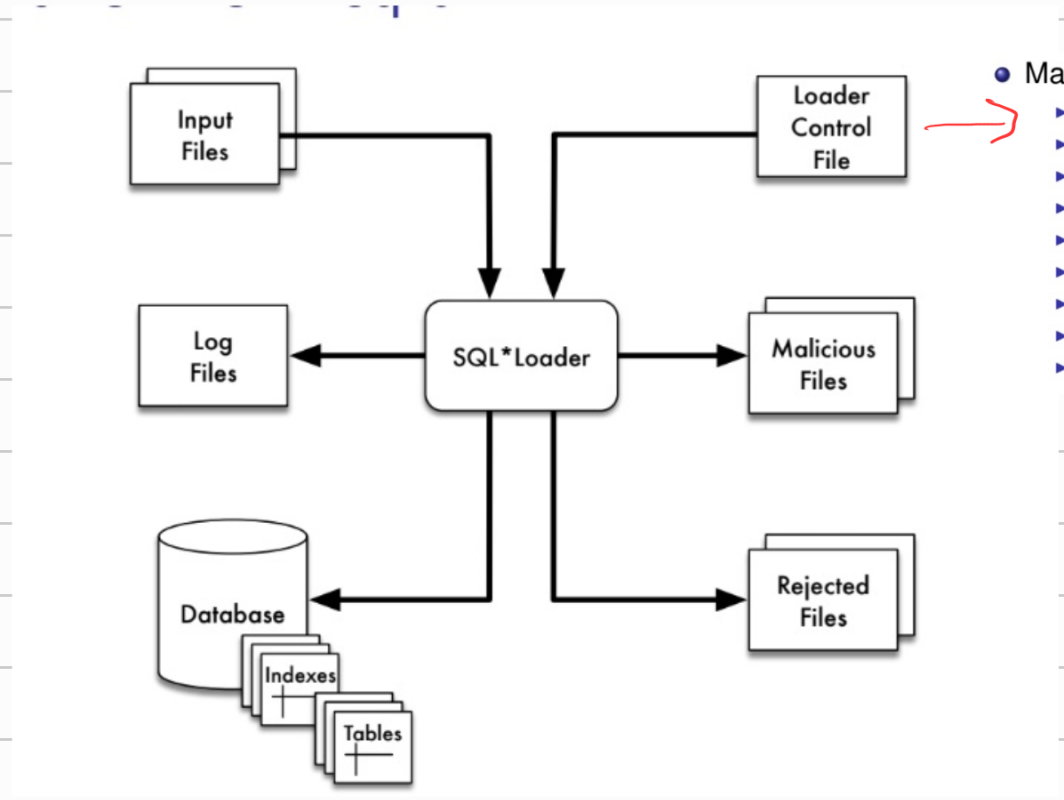
Advantages:

- **Faster loading speed** compared to row-by-row inserts.
- **Efficient use of system resources** since it minimizes transaction logging.

Use Cases:

- Initial data warehouse population.
- Periodic data updates from external systems (e.g., **daily or weekly batch ETL processes**).

Oracle Sqlldr:



Transactional Loading → integrity checks on each transaction

Definition: A method where data is inserted into the Data Warehouse as individual transactions in real time or near real time.

Characteristics:

- Uses **row-by-row insertion** rather than bulk operations.
- Each row is processed as a separate transaction.
- Maintains **ACID (Atomicity, Consistency, Isolation, Durability)** properties.
- Typically implemented using **ETL tools (e.g., Talend, Informatica, Apache Kafka)** for streaming data.

Advantages:

- **Ensures real-time data availability** for operational dashboards.
- **Better data consistency** as transactions are logged immediately.

Use Cases:

- **Real-time analytics** (e.g., monitoring stock prices, fraud detection).
- **OLTP databases feeding into a Data Warehouse**.

Differential SnapShot Problem:

Problem: • Many Sources provide only the full dataset
Eg: Molecular biology databases
Product catalogues
• Repeated import of all data is inefficient
• Duplicates need to be detected

Scenario

- Sources provide Snapshots as file F
  - Unordered set of records $(K, A_1, \ldots, A_n)$
- Given: F_1, F_2, mit $f_1 = |F_1|, f_2 = |F_2|$
- Calculate smallest set $O = \{\textbf{INS}, \textbf{DEL}, \textbf{UPD}\}^*$ with $O(F_1) = F_2$
- O not unique!

$$O_1 = \{(\textbf{INS}(X)), \emptyset, (\textbf{DEL}(X))\} \equiv O_2 = \{\emptyset, \emptyset, \emptyset\}$$

Check exercise 5:

DS naive → Nested Loop.

DS small → Small files

DS sort → Sort Merge

DS sort 2 → Interleaved

DS hash → Partitioned Hash

Comparison – Features

	IO	Bemerkungen
DS_{naive}	$f_1 \cdot f_2$	out of concurrence, auxiliary data structure required
DS_{small}	$f_1 + f_2$	only for smaller files
DS_{sort2}	$f_1 + 4 \cdot f_2$	
DS_{hash}	$f_1 + 3 \cdot f_2$	non-overlapping hash function, hard to estimate partition size, assumptions about distribution (Sampling)

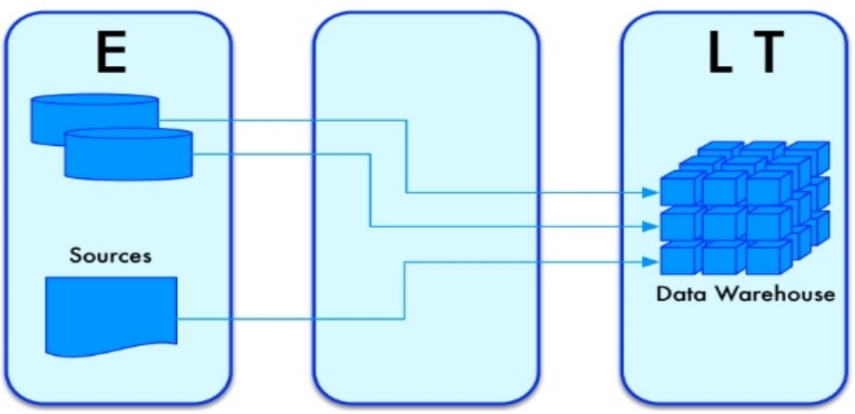
- Extensions of DS_{hash} for "worse" hash functions known

ELT

- Extraction
  - For Database optimized queries (e.g. SQL)
  - Extraction also monitored with monitors
  - Automatic extraction difficult (e.g. data structure changes)
- Laden
  - Parallel processing of SQL statements
  - Bulk Load (assumption: no write access to the target system)
  - No record-based logging
- Transformation
  - Utilization of set operations of the DW-transformation component
  - Complex transformations by means of procedural languages (e.g., PL/SQL)
  - Specific statements (e.g., CTAS von Oracle)

ETL vs. ELT

- ELT = Extract-Load-Transform
  - Variant of the ETL process, in which the data is transformed after the load
  - Objective: transformation with SQL statements in the target database
  - Waiving special ETL engines



4) Duplicate Detection: (X) (Y)

multipass technique

check Exercise 6.

5) Transformation on Example

check Exercise 6

6) Identify Data Quality issues via SQL

Duplicate Detection, Multi-Pass Technique, and Important Aspects (Based on Lecture Slides)

1. Duplicate Detection

Duplicate detection is the process of identifying and eliminating redundant records in a data warehouse to ensure data integrity and accuracy.

Key aspects:

- **Semantic equivalence:** Two records refer to the same real-world entity but might have slight variations in spelling, format, or structure.
- **Common techniques:**
  - **Record Linkage:** Identifies duplicate records by comparing attributes like name, address, and ID.
  - **Object Identification:** Matches entities across different datasets.
  - **Merge & Purge:** Combines duplicate records while selecting the best representative per class.

Example from Slides:

CustomerNr	Name	Address
3346	Just Vorfan	Hafenstrasse 12
3346	Justin Forfun	Hafenstr. 12
5252	Lilo Pause	Kuhweg 42
5268	Lisa Pause	Kuhweg 42
?	Ann Joy	Domplatz 2a
?	Anne Scheu	Domplatz 28

This example highlights how minor differences in names and addresses can cause duplicate records .

---

2. Multi-Pass Technique

The **multi-pass technique** is a strategy to improve duplicate detection by applying multiple sorting and matching steps with different keys.

How it works:

1. **Sort records based on one criterion (e.g., name).**
2. **Identify duplicates within a sorted window.**

3. **Re-sort using another criterion (e.g., address).**
4. **Identify additional duplicates missed in the first pass.**
5. **Form a transitive closure to merge related duplicates.**

Example: 1st Pass: "A matches B"
2nd Pass: "B matches C"
Result (Transitivity Rule): "A matches C"

Advantage: Increases accuracy in detecting duplicates that were missed in a single-pass approach .

---

3. Important Things in Duplicate Detection

- **Comparison Functions:**
  - **Edit Distance (Levenshtein Distance):** Measures the number of changes needed to transform one string into another (e.g., "Qualität" → "Quantität" has an edit distance of 2).
  - **q-Grams:** Breaks strings into smaller substrings and compares overlap.
  - **Jaro-Winkler Distance:** Considers common characters and transpositions.
- **Partitioning Techniques:**
  - **Blocking:** Divides data into smaller groups, comparing only within each block.
  - **Sorted Neighborhood:** Sorts records by a key (e.g., SSN, name) and compares them within a sliding window .

Summary

- **Duplicate detection** identifies semantically equivalent records and eliminates redundancy.
- **Multi-pass technique** improves detection by applying multiple sorting and matching steps.
- **Key techniques** include edit distance, q-grams, and Jaro-Winkler comparisons.
- **Partitioning (blocking, sorted neighborhood)** helps optimize performance.

OLAP Queries

1) CUBE :-

CUBE and ROLLUP (Based on Lecture Slides)

CUBE and ROLLUP are SQL operations used in **OLAP (Online Analytical Processing)** to aggregate data at different levels of granularity. These functions help in generating **multi-dimensional reports** by computing subtotals and grand totals across different dimensions.

---

1. CUBE Operator → inter dimensional with an example single dimn & concat dims.

Definition:

The **CUBE** operator generates **all possible combinations of groupings** for a given set of attributes, providing subtotals for every possible grouping along with the **grand total**.

Key Features:

- Produces 2^n **grouping combinations** for **n** attributes. (4 attributes → 16 combinations)
- Useful for **multi-dimensional analysis**.
- Generates **all possible aggregations**, making it suitable for **pivot tables**.

Example (From Lecture Slides)

Consider a sales dataset with **Product Group (PGroup), Year, and Region**, and we want to analyze turnover.

Query using CUBE:

sql Copy

```
SELECT P_ProductGroup AS PGroup, L_Region, YEAR(T_Date), SUM(S_Number * P_SalesPrice) AS Turnover FROM Sale, Time, Product, Location GROUP BY CUBE(P_ProductGroup, L_Region, YEAR(T_Date));
```

Result:

PGroup	Year	Region	Turnover
Wine	2010	Saxony-Anhalt	45
Wine	2010	Thuringia	43
Wine	2010	NULL	88
Wine	NULL	NULL	135

PGroup	Year	Region	Turnover
NULL	NULL	NULL	177

- **NULL values** represent **aggregated totals** at different levels.
- The **last row** (NULL, NULL, NULL) contains the **grand total**.

---

2. ROLLUP Operator → intra dimensional

Definition:

The **ROLLUP** operator is used to generate **hierarchical aggregations**, meaning it produces subtotals **only for a single hierarchy of attributes** rather than all possible combinations.

Key Features:

- Computes **cumulative subtotals** moving up the hierarchy.
- Produces **n+1** grouping combinations for **n** attributes (less than CUBE).
- Suitable for **reporting hierarchical data** (e.g., Year → Quarter → Month).

Example (From Lecture Slides)

Consider a sales dataset where we want to calculate **daily, monthly, and yearly turnover** for a product.

Query using ROLLUP:

sql Copy

```sql
SELECT P_group, T_day, T_month, T_year, SUM(S_Number * P_SalesPrice) AS Turnover FROM Sale, Time, Product, Place WHERE S_Product_ID = P_ID AND S_Time_ID = T_ID AND YEAR(T_Date) = 2011 AND P_ProductGroup = 'Red Wine' GROUP BY ROLLUP(T_year, T_month, T_day);
```

Result:

Product Group	Day	Month	Year	Turnover
Red Wine	1	Jan	2011	100
Red Wine	2	Jan	2011	100
Red Wine	NULL	Jan	2011	2000

Product Group	Day	Month	Year	Turnover
Red Wine	NULL	NULL	2011	24000
Red Wine	NULL	NULL	NULL	24000

- The **NULL values** show the aggregated turnover at **monthly, yearly, and grand total levels**.

---

3. CUBE vs. ROLLUP – Key Differences

Feature	CUBE	ROLLUP
Grouping Combinations	All 2^n combinations	Only **hierarchical** (n+1) aggregations
Use Case	Full OLAP cube for multi-dimensional analysis	Hierarchical aggregations (e.g., Year → Month → Day)
Performance	More expensive (larger result set)	More optimized for hierarchy-based summaries
Example Output	Produces all subtotals and totals at different levels	Produces only hierarchical totals

---

4. Summary

- **CUBE** is useful when you need **all possible aggregations**, including cross-dimensional subtotals.
- **ROLLUP** is efficient for **hierarchical aggregations** where totals are computed **step-by-step**.

Grouping Sets, OVER Clause, and Special Aggregate Functions (Based on Lecture Slides)

---

1. GROUPING SETS

Definition:

- **GROUPING SETS** allow for customized **multiple groupings in a single query**, improving flexibility and efficiency compared to using multiple `GROUP BY` statements with `UNION ALL`.

Key Features:

- Computes **specific** subtotal combinations, unlike `ROLLUP` (hierarchical) and `CUBE` (all combinations).
- **Performance-optimized**—only the required aggregations are computed.

Example Query (From Lecture Slides)

sql Copy

```sql
SELECT P_ProductGroup, L_City, L_Province, YEAR(T_Date), SUM(S_Number * P_SalesPrice) AS Turnover FROM Sale, Time, Product, Place WHERE S_Time_ID = T_ID AND S_Product_ID = P_ID GROUP BY GROUPING SETS ( (L_City), (L_Province), (ROLLUP(YEAR(T_Date), QUARTER(T_Date), MONTH(T_Date))) );
```

Interpretation:

- `(L_City)` : Aggregation per city.
- `(L_Province)` : Aggregation per province.
- `(ROLLUP(YEAR, QUARTER, MONTH))` : Aggregates **yearly** → **quarterly** → **monthly** sales.

Comparison:

Feature	GROUPING SETS	ROLLUP	CUBE
Grouping Strategy	Custom-defined sets	Hierarchical	All combinations
Number of Aggregations	Limited	n+1 levels	2^n groupings
Best Use Case	Selective aggregations	Hierarchical reports	Full OLAP analysis

2. OVER Clause

Definition:

- The `OVER()` clause is used with **window functions** to perform calculations **across a set of rows** related to the current row.
- It enables **ranking, running totals, and moving averages without** collapsing the result set.

Key Features:

- Defines **partitioning and ordering** for aggregations.
- Computes **cumulative sums, moving averages, ranking, and percentiles**.

Example Query (From Lecture Slides)

sql Copy

```sql
SELECT T_Date, Turnover, SUM(Turnover) OVER() AS MonthTotal, 100.0 * Turnover /
SUM(Turnover) OVER() AS Share FROM DailyTurnover WHERE P_ProductGroup = 'Wine' AND
YEAR_MONTH(T_Date) = 201108;
```

Breakdown:

- `SUM(Turnover) OVER()` → Computes total turnover **across all rows**.
- `100.0 * Turnover / SUM(Turnover) OVER()` → Computes **percentage contribution**.

Partitioning and Ordering with OVER Clause

Partitioning:

- Divides the dataset into subsets for aggregation **without reducing the number of rows**.

sql Copy

```sql
SUM(Turnover) OVER(PARTITION BY YEAR_MONTH(T_Date))
```

- This ensures aggregation is done **per month** rather than across the entire dataset.

Ordering:

- Used for **cumulative totals, running averages, and rankings**.

```sql
SUM(Turnover) OVER(ORDER BY T_Date)
```

- This provides a **progressive sum** of turnover over time.

---

3. Special Aggregate Functions

SQL:2003 introduced **advanced statistical and analytical functions**:

Function	Purpose
`VAR_POP(x)`	Population variance
`STDDEV_POP(x)`	Population standard deviation
`COVAR_POP(x, y)`	Covariance between two variables
`CORR(x, y)`	Pearson correlation coefficient
`REGR_SLOPE(y, x)`	Regression line slope
`REGR_R2(y, x)`	Coefficient of determination (R^2)
`REGR_AVGX(y, x)`	Mean of independent variable ($\bar{x}$)
`REGR_AVGY(y, x)`	Mean of dependent variable ($\bar{y}$)

Example: Correlation Analysis

Objective: Find correlation between product price and quantity sold.

```sql
SELECT CORR(S_Number, P_SalesPrice) AS Correlation FROM Sales, Product WHERE S_Product_ID
= P_ID;
```

Interpretation:

- **Positive correlation (> 0.5)**: Higher price leads to higher sales.
- **Negative correlation (< -0.5)**: Higher price reduces sales.
- **Close to 0**: No significant correlation.

---

4. Summary

Feature	Definition	Use Case
GROUPING SETS	Custom aggregations	Selective subtotaling
OVER Clause	Enables window functions	Cumulative totals, rankings
Special Aggregate Functions	Advanced statistical analysis	Correlation, variance, regression

3) Iceberg cube:

Based on given example create Iceberg cube queries.

Iceberg Cube and Iceberg Queries (Based on Lecture Slides)

---

1. What is an Iceberg Cube?

Definition:

An **Iceberg Cube** is an **optimized version of the CUBE operator** that calculates **only necessary aggregations** instead of computing all possible groupings. It **filters out insignificant aggregations** by applying a **HAVING condition** to limit the number of computed groups.

Why is it Needed?

- The standard **CUBE operator** computes **all** possible group-by combinations, which can lead to **huge data expansion**.
- Many of these groups contain **few or no records**, wasting computation time and storage.
- Iceberg Cube **prunes unnecessary aggregations** by computing only those **meeting a minimum support threshold (N)**.

---

2. Example: Why Use Iceberg Cube?

Consider a **weather dataset** with **9 dimensions**:

- The full **CUBE operation** generates **210,343,580 rows** (≈200 times the original input size).
- However, **only 50 times the input size** contains at least **2 records**, and just **5 times** contains at least **10 records**.
- Most groupings are **too small to be meaningful**, so Iceberg Cube **filters them out**.

---

3. How to Create an Iceberg Query?

Syntax:

sql Copy

```
SELECT A, B, C, COUNT(*), SUM(X) FROM R GROUP BY CUBE(A, B, C) HAVING COUNT(*) >= N;
```

Explanation:

- `CUBE(A, B, C)` : Computes all possible groupings of attributes A, B, and C.
- `HAVING COUNT(*) >= N` : **Filters out** groups with fewer than `N` records.

---

4. Example: Iceberg Query for Sales Data

Scenario:
We want to compute the total **sales turnover** at different levels of aggregation but **only keep aggregations with at least 100 transactions**.

SQL Query:

sql Copy

```
SELECT P_ProductGroup, L_Region, YEAR(T_Date), SUM(S_Number * P_SalesPrice) AS Turnover,
COUNT(*) AS TransactionCount FROM Sale, Time, Product, Location GROUP BY
CUBE(P_ProductGroup, L_Region, YEAR(T_Date)) HAVING COUNT(*) >= 100;
```

Interpretation:

- **Removes small, irrelevant groupings** where transaction count is below 100.
- Ensures only **meaningful aggregates** are stored in the Data Warehouse.

---

5. Optimization Techniques for Iceberg Cube

- **Pruning:** Stops processing for groupings that do not meet the threshold.
- **Early Aggregation:** Computes aggregates **bottom-up**, avoiding unnecessary calculations.
- **Indexing:** Uses **pre-computed summaries** for faster retrieval.

---

6. Summary

Feature	Standard CUBE	Iceberg CUBE
Computes all aggregations?	✅ Yes	❌ No
Filters out low-support groups?	❌ No	✅ Yes
Performance Efficiency	🔴 High Cost	🟢 Optimized
Use Case	Small datasets	Large-scale OLAP queries

Final Takeaway:

- **Iceberg Cube** is ideal for **big data scenarios** where full cube computation is too expensive.
- It **keeps only the relevant aggregations** based on a **minimum support threshold**.

Storage Structures

check Exercise 8 for Dwarf creation (X) for exam.

1. Partitioning and Materialization Strategies

Partitioning

Partitioning is a method to **divide large database tables** into smaller, more manageable parts. This improves **query performance, manageability, and parallel processing** in a Data Warehouse.

Types of Partitioning:

1. **Horizontal Partitioning:**
   - The table is divided into **sub-tables (partitions)** based on row values.
   - **Example:** A sales table is split by **years** (Sales_2020, Sales_2021, etc.).
   - **Methods:**
     - **Range Partitioning:** Uses a range of values (e.g., Date < '2023-01-01').
     - **Hash Partitioning:** Uses a hash function (e.g., partitioning customers based on their ID).
2. **Vertical Partitioning:**
   - The table is split into **sub-tables based on columns**.
   - **Example:** A **customer table** may have separate partitions for **basic details (name, address)** and **financial details (credit score, spending history)**.

Materialization Strategies

Materialization refers to **storing precomputed query results** for faster access. There are two key strategies:

1. **Early Materialization:**
   - Tuple reconstruction occurs **early in query execution**.
   - **Advantage:** Works well for **row-oriented** databases.
   - **Disadvantage:** Can cause performance bottlenecks.
2. **Late Materialization:**
   - Data is processed in **columnar format** as long as possible.
   - **Advantage:** Efficient for **column-oriented databases**.
   - **Disadvantage:** Requires **additional joins** at the final stage.

---

2. ROLAP vs. MOLAP vs. HOLAP

Feature	ROLAP (Relational OLAP)	MOLAP (Multidimensional OLAP)	HOLAP (Hybrid OLAP)
Storage	Relational database tables (Star Schema)	Multidimensional cube	Combination of both
Data Retrieval Speed	Slower due to SQL queries	Faster due to precomputed cubes	Balances performance
Query Performance	Good for detailed queries	Excellent for pre-aggregated data	Adapts based on usage
Data Scalability	Best for large datasets	Limited scalability	Medium scalability
Storage Efficiency	Requires large storage	Compressed storage	Optimized storage
Example Use Case	**Ad-hoc queries** and detailed reporting	**Predefined reports** for executives	**Balanced approach** for mixed workloads

---

3. Data Dwarf and How to Create It

What is Data Dwarf?

A **Data Dwarf** is a **highly compressed** version of a multidimensional cube. It removes redundant values using **prefix and suffix compression**.

Key Features:

- Reduces **storage size drastically**.
- Uses **hierarchical compression** for dense and sparse data.
- Suitable for **mobile networks and distributed computing**.

Example:

Consider the following **Sales Data Cube:**

Region	Customer	Product	Price
R1	C2	P2	70
R1	C3	P1	40
R2	C1	P1	90
R2	C1	P2	50

After applying **Dwarf Compression**, redundant prefixes and suffixes are **merged**, reducing memory usage.

Steps to Create a Dwarf:

1. Identify **prefix and suffix redundancies** in hierarchical dimensions.
2. Apply **hierarchical merging** to remove redundant storage.
3. Store **compressed results** for efficient retrieval.

---

4. Column Stores vs. Row Stores

For query execution & how are they fit for OLAP

Feature	Row Store	Column Store
Storage Method	Data stored row-wise (tuples)	Data stored column-wise
Read Performance	Slower for aggregations	Faster for aggregations (OLAP)
Write Performance	Faster for transactional updates	Slower due to columnar nature
Compression	Less efficient	Highly efficient (10:1 to 40:1)
Best Use Case	OLTP (e.g., banking, retail)	OLAP (e.g., Data Warehousing)
Example Databases	MySQL, PostgreSQL, Oracle	Apache Cassandra, Amazon Redshift

---

Final Summary

- **Partitioning** improves **query speed** by dividing data logically.
- **Materialization strategies** optimize query execution.
- **ROLAP, MOLAP, and HOLAP** serve different **data processing needs**.
- **Data Dwarf** significantly **compresses** multidimensional cubes.
- **Column stores** outperform **row stores** for **analytics**, but **row stores** are better for **transactions**.

Index Structures

Classification of index structures

- Clustering: data that are likely to often processed together will also be stored physically close to each other
  - *Tuple Clustering*: Storing of tuples on the same physical page
  - *Page Clustering*: storing related pages close together in secondary storage (allows *prefetching*)
- Dimensionality: specify how many attributes (dimensions) of the underlying relation for calculation of the index key can be used
- Symmetry: If the performance is independent of the order of the index attributes, we have a symmetrical index structure, otherwise asymmetrical
- Tuple references: Type of tuple references within the index structure
- Dynamic behavior: Effort to update the index structure for insert, update and delete; (and possibly problem of "degeneration")

What is symmetry & apply it for index structures

1. Characterization of Index Structures

(Referenced from Slide 7-3, 7-4)

Index structures in data warehouses are designed to enhance query performance by minimizing the number of required disk accesses. The classification of index structures includes:

1.1 Clustering

- **Tuple Clustering:** Tuples that are often accessed together are stored on the same physical page.
- **Page Clustering:** Related pages are stored close together in secondary storage, allowing prefetching.

1.2 Dimensionality

- Defines the number of attributes (dimensions) used for calculating the index key.
- Higher-dimensional indexes are necessary for multidimensional queries in OLAP (Online Analytical Processing).

1.3 Symmetry

- **Symmetrical Index:** Performance is independent of the order of index attributes.
- **Asymmetrical Index:** Performance is affected by the order of attributes in the index.

1.4 Tuple References

- Specifies how tuples are referenced within the index structure (e.g., direct TID references).

1.5 Dynamic Behavior

- Measures the effort needed to update the index upon inserts, updates, or deletions.
- Some structures, like **B-Trees**, maintain balance dynamically, while others (e.g., bitmap indexes) are more static.

---

2. B-Tree Specialties

(Referenced from Slides 7-6 to 7-13)

2.1 Definition and Properties

- A **B-Tree** is a balanced, one-dimensional tree structure used for indexing.

- Each node can contain at most **2m** elements and at least **m** elements (except the root).
- All leaf nodes are at the same level, ensuring balanced performance.

2.2 Concatenation of Column Values

- **Multi-column indexing** can be implemented by concatenating multiple attribute values.
- Example:

```sql
CREATE INDEX idx ON Customer(cclass, gender, profession);
```

  - Allows efficient searching for queries like:

```sql
SELECT * FROM Customer WHERE cclass=1 AND gender='m' AND profession='Lecturer';
```

  - However, the order of indexed attributes matters (order-sensitive).

2.3 Symmetry in B-Trees

- **Balanced Structure:** Ensures logarithmic search time **O(log n)**.
- **Efficient for range queries** because leaf nodes are linked sequentially (**B+-Tree variant**).
- **Issues with Low Cardinality:** Attributes with few distinct values (e.g., gender) can lead to **degenerated trees**.

---

3. Grid File

(Referenced from Slides 7-44 to 7-48)

3.1 Definition

- A **Grid File** is a multidimensional indexing structure combining **hashing and tree-based indexing**.
- It divides multidimensional space into **grid cells** to efficiently organize data.

3.2 Working Principle

- Uses **equal distribution (hyperplanes)** to partition dimensions.
- **Grid Directory:** Contains grid cells that map to data records.

- **Grid Cells:** Group similar data into **searchable regions**.

3.3 Advantages

- Efficient for **partial match queries** in multidimensional space.
- Supports **dynamic updates** without major reorganization.

---

4. Bitmap Indexes

(Referenced from Slides 7-20 to 7-36)

4.1 Definition

- Uses **bitmaps** to encode attribute values for efficient filtering and joins.
- Particularly useful for **low-cardinality attributes** (e.g., gender, status).

4.2 Encoding and Example

Given a **Gender** attribute with values {Male, Female}:

PersId	Name	Gender	Bitmap-F	Bitmap-M
007	James Bond	M	0	1
008	Amelie Lux	F	1	0
010	Harald Schmidt	M	0	1
011	Heike Drechsler	F	1	0

- **Number of Bitmaps Needed:** One per distinct value.
  - In this case, **two bitmaps** (one for Male, one for Female).
- **Query Example:** Find all Female customers

```sql
SELECT * FROM Customer WHERE Gender='F';
```

  - The system performs a **bitwise AND** on the **Bitmap-F** vector.

4.3 Advantages

- **Storage-efficient** for read-heavy workloads.

- **Fast filtering operations** using **bitwise operations**.
- **Well-suited for Star Schema joins** in data warehouses.

---

5. UB-Tree

(Referenced from Slides 7-66 to 7-72)

5.1 Definition

Z-ordering (Morton order)

- A **UB-Tree (Universal B-Tree)** is a multidimensional indexing structure using **Z-order curves**.
- It transforms multidimensional data into **one-dimensional space** for indexing.

5.2 How It Works

1. **Convert attributes into binary representations.**
2. **Interleave bits** from different dimensions to form a **Z-value**.
3. **Index Z-values using a B+-Tree.**

B+-Tree

5.3 Example

Consider two-dimensional data:

```scss
(X, Y) → Binary Encoding → Z-Value
(2,3) → (10, 11) → 1011
(4,5) → (100, 101) → 11001
```

Copy Edit

- Data is stored based on **Z-order values**, ensuring spatial locality.

5.4 Benefits

- Efficient for **range queries**.
- Compact structure using **B+-Tree indexing**.

---

6. R-Tree

(Referenced from Slides 7-57 to 7-60)

6.1 Definition

- **R-Trees (Rectangle Trees)** are designed for **spatial indexing**.
- They use **minimum bounding rectangles (MBRs)** to group spatial objects.

6.2 How It Works

Used in -> GIS, Game Engines & Databases to quickly find objects within a specific area.

1. **Each node stores an MBR** that encloses child nodes.
2. **Search operation** checks for MBR overlaps.
3. **Insertions expand MBRs** dynamically as needed.

6.3 Example

Consider a **map with different regions**:

```diff
- R1: City Blocks (A, B, C)
- R2: Parks (D, E, F)
- R3: Lakes (G, H)
```

- A **query to find all objects in region R2** checks **MBR overlaps**.

6.4 Advantages

- Handles **geographical queries** efficiently.
- Supports **range queries and spatial joins**.

Summary

Index Type	Best Use Case	Advantages
B-Tree	One-dimensional indexing	Balanced, efficient for range queries
Grid File	Multi-dimensional queries	Fast lookups, dynamic storage
Bitmap Index	Low-cardinality attributes	Space-efficient, fast filtering
UB-Tree	Multi-dimensional range queries	Uses Z-order for efficient storage

Index Type	Best Use Case	Advantages
R-Tree	Spatial data indexing	Optimized for geometric searches

---

These explanations are based on the **lecture slides** and provide a comprehensive understanding of the index structures in data warehousing. Let me know if you need further elaboration! 🚀

Query Optimization

Refer Exercise 10 (x?)

Before exam again.

Star Join in Query Optimization

Definition

A **Star Join** is a specialized join strategy used in **data warehousing** to optimize queries on **star schema** structures. It significantly improves query performance by reducing the number of required table scans and joins.

Context: Star Schema

In a **star schema**, we have:

- A **large fact table** containing transactional data.
- **Multiple smaller dimension tables** that provide descriptive attributes for the facts.

Example: **Retail Sales Schema**

sql Copy Edit

```
Fact Table: Sales (Fact)
+-----------+------------+------------+----------+-------+
| Sales_ID  | Time_ID    | Store_ID   | Prod_ID  | Sales |
+-----------+------------+------------+----------+-------+

Dimension Tables:
1. Time (Time_ID, Year, Month, Day)
2. Store (Store_ID, Location, Region)
3. Product (Prod_ID, Name, Category)
```

A typical query in a star schema:

sql Copy Edit

```
SELECT SUM(s.Sales), t.Year, p.Category FROM Sales s JOIN Time t ON s.Time_ID =
t.Time_ID JOIN Product p ON s.Prod_ID = p.Prod_ID WHERE t.Year = 2023 AND p.Category =
'Electronics' GROUP BY t.Year, p.Category;
```

How Star Join Optimizes Queries

In a **traditional join** approach:

- Each dimension table is **joined separately** with the fact table, causing **multiple full table scans**.
- Query performance degrades with increasing data size.

In a **Star Join** approach:

- **Bitmap Indexes & Bitmap Join Indexes** are used to efficiently filter records in the fact table.
- The **fact table is accessed first**, and **dimension table filtering is applied later**.
- This reduces the number of rows to scan **before performing joins**.

Implementation Techniques

1. **Bitmap Index on Dimension Keys in Fact Table**
   - Example:

```sql
CREATE BITMAP INDEX time_idx ON Sales(Time_ID); CREATE BITMAP INDEX product_idx ON Sales(Prod_ID);
```

- Enables **fast filtering** using bitwise operations.

2. **Bitmap Join Index**
   - Precomputes joins between the fact and dimension tables.
   - Example:

```sql
CREATE BITMAP INDEX join_idx ON Sales(Product.Category) FROM Sales, Product WHERE Sales.Prod_ID = Product.Prod_ID;
```

   - Optimizes queries that frequently filter by product category.
3. **Join Indexing Strategy**
   - Instead of **joining all dimension tables first**, the optimizer:
     1. **Filters the fact table using bitmap indexes**.
     2. **Joins the filtered fact table with dimension tables** (instead of scanning the entire fact table).

Example of Star Join Execution Plan

1. **Filter the fact table using bitmap indexes**

```sql
SELECT * FROM Sales WHERE Time_ID = 2023 AND Prod_ID IN (SELECT Prod_ID FROM Product WHERE Category='Electronics');
```

2. **Perform the actual join after filtering**

```sql
SELECT SUM(Sales) FROM (Filtered Fact Table) s JOIN Time t ON s.Time_ID = t.Time_ID JOIN Product p ON s.Prod_ID = p.Prod_ID
```

Advantages of Star Join Optimization

✅ **Minimizes Scans:** Fact table is filtered before join execution.
✅ **Reduces Join Overhead:** Uses bitmap indexes to efficiently eliminate rows.
✅ **Improves Query Speed:** Optimized for **OLAP queries** with **high selectivity**.
✅ **Ideal for Star Schema Queries:** Reduces **full table scans** and **hash join operations**.

Conclusion

A **Star Join** is a crucial optimization technique for **data warehouse queries**, especially when dealing with **large fact tables** and **multi-dimensional analysis**. It leverages **bitmap indexing** and **precomputed joins** to speed up query execution.

Would you like an **SQL example** demonstrating how to implement a **Star Join with bitmap indexes**? 🚀

Also explain Invariant grouping and Early Pre-groupin in Query optimization

Invariant Grouping & Early Pre-Grouping in Query Optimization

In **query optimization**, grouping operations (like `GROUP BY` and `AGGREGATE FUNCTIONS`) can be computationally expensive, especially in **OLAP (Online Analytical Processing)** workloads. Two key optimization strategies to improve performance are **Invariant Grouping** and **Early Pre-Grouping**.

---

1. Invariant Grouping

Definition

Invariant Grouping is a query optimization technique that **removes unnecessary attributes from the grouping step**, ensuring that only relevant attributes are used for aggregation.

Concept

- **Groupings that do not change (invariant attributes) across a query execution can be factored out.**
- **Redundant columns** in `GROUP BY` clauses are removed before executing the grouping operation.
- The optimizer **detects dependencies** between grouped attributes and eliminates redundant ones.

Example

Query Without Optimization

```sql
SELECT region, city, SUM(sales) FROM Sales GROUP BY region, city;
```

- Here, `region` and `city` are grouped.
- If **each city belongs to only one region**, then `region` is **functionally dependent** on `city`.

Optimized Query Using Invariant Grouping

```sql
SELECT city, SUM(sales) FROM Sales GROUP BY city;
```

- Since **region is uniquely determined by city**, we can eliminate `region` from the `GROUP BY` clause.
- This **reduces the number of grouping keys**, leading to **faster aggregation**.

Benefits of Invariant Grouping

✅ **Reduces unnecessary computations** by eliminating redundant columns.
✅ **Speeds up query execution** by lowering the number of groups.
✅ **Decreases memory usage** during grouping and aggregation.

---

2. Early Pre-Grouping

Definition

Early Pre-Grouping is an optimization technique that performs **partial aggregation as early as possible** in the query execution plan, before the final grouping step.

Concept

- Instead of aggregating all records at once, **pre-grouping is done at an earlier stage** (e.g., at the scan level or before joins).
- This reduces **intermediate result size** and speeds up processing.

Example

Query Without Optimization

```sql
SELECT product_category, SUM(sales) FROM Sales JOIN Product ON Sales.product_id = Product.product_id GROUP BY product_category;
```

- The **JOIN** is performed **before aggregation**, causing a large intermediate table.
- If `Sales` contains **millions of rows**, the join **explodes in size**, slowing down the query.

Optimized Query Using Early Pre-Grouping

```sql
SELECT product_category, SUM(total_sales) FROM ( SELECT product_id, SUM(sales) AS total_sales FROM Sales GROUP BY product_id ) AS PreGrouped JOIN Product ON PreGrouped.product_id = Product.product_id GROUP BY product_category;
```

What Happens Here?

1. **Step 1: Pre-grouping on `Sales` table**
   - `SUM(sales)` is calculated **before the join**, reducing the number of rows significantly.
2. **Step 2: Join with `Product` table**
   - The join now works on a **smaller dataset**.
3. **Step 3: Final Aggregation on `product_category`**
   - Aggregation is performed on a **much smaller intermediate table**.

Benefits of Early Pre-Grouping

✅ **Drastically reduces intermediate table size** before joins.
✅ **Optimizes performance for large fact tables in data warehouses.**
✅ **Reduces memory consumption** during query execution.

Comparison: Invariant Grouping vs. Early Pre-Grouping

Optimization Technique	Purpose	How It Works	Example Benefit
Invariant Grouping	Eliminates redundant `GROUP BY` attributes	Removes functionally dependent attributes	Reduces number of groups, speeds up aggregation
Early Pre-Grouping	Performs early aggregation before joins	Reduces intermediate data size	Avoids large join explosion, speeds up query execution

---

Conclusion

- **Invariant Grouping** simplifies `GROUP BY` operations by eliminating unnecessary attributes.
- **Early Pre-Grouping** performs aggregation early, reducing intermediate data size and improving join performance.
- Both techniques are **critical for query optimization in data warehouses**, especially for **large-scale OLAP workloads**.

Would you like **a real-world SQL query example** demonstrating both techniques with an **execution plan**? 🚀

What is Data cube lattice and Pipesort and how to create data cube lattice and Pipesort

1. Data Cube Lattice

Definition

A **Data Cube Lattice** represents all possible aggregations (grouping and summarization) of a dataset across multiple dimensions. It is a structure that organizes the different levels of aggregation in an OLAP **data cube**, allowing efficient query processing and summarization in **data warehousing**.

1.1 How a Data Cube Lattice Works

- A **data cube** aggregates data across multiple dimensions (e.g., Time, Product, Region).
- The **lattice** shows all possible combinations of aggregations.
- Each node in the **lattice** represents a different level of summarization.
- The **top node** (apex) represents full aggregation (`ALL` values).
- The **bottom node** represents the raw data (no aggregation).

1.2 Example: Data Cube with 3 Dimensions

Consider a dataset with **3 dimensions**:

- **Time**
- **Product**
- **Region**

The **Data Cube Lattice** for these 3 dimensions will have **8 aggregations** ($2^3 = 8$):

```mathematica
                (ALL)  → Total sales (Fully Aggregated)
               /   |   \
          Time  Product  Region  → Aggregated over one dimension
         /  \     /  \    /  \
Time-Product Time-Region Product-Region  → Aggregated over two dimensions
          \       |       /
       Time-Product-Region  → No aggregation (Raw Data)
```

Each node represents a level of aggregation, and edges represent derivation paths.

1.3 SQL Query to Compute a Data Cube Lattice

To compute all aggregations, we use the `CUBE` operator:

```sql
SELECT Time, Product, Region, SUM(Sales) AS Total_Sales FROM Sales GROUP BY CUBE(Time, Product, Region);
```

This generates all $2^3 = 8$ possible groupings in the lattice.

---

2. Pipesort Algorithm

2.1 Definition

Pipesort is an **incremental sorting algorithm** used for efficiently sorting and aggregating large data sets **without requiring multiple passes over the data**. It is particularly useful for **OLAP cube computation** and **query optimization**.

2.2 How Pipesort Works

- **Sort data incrementally** as it arrives in a pipeline fashion.
- **Merge partial results** dynamically instead of sorting the entire dataset at once.
- **Reduce I/O and memory usage** by performing sorting and aggregation in a single pass.
- **Optimized for large-scale databases** where full in-memory sorting is not feasible.

---

3. Creating a Data Cube Lattice with Pipesort

To compute a **Data Cube** efficiently using **Pipesort**, we follow these steps:

Step 1: Load Data in Sorted Order

- Sort data **incrementally** based on the dimensions.
- Example:

```sql
SELECT Time, Product, Region, SUM(Sales) FROM Sales ORDER BY Time, Product, Region;
```

Step 2: Apply Pipesort to Aggregate Data

- Process each incoming record **incrementally**.

- Maintain **sorted buffers** for different levels of aggregation.
- Combine and output results dynamically.

Step 3: Compute Aggregations in Lattice Order

- Instead of computing all groupings at once, **use partial results from previous computations**.
- Example:
  - First, compute `(Time, Product, Region)`.
  - Then, derive `(Time, Product)`, `(Time, Region)`, etc., using incremental processing.

Step 4: Store or Output the Cube

- Write the results to a **materialized view** or return them in a query.

---

4. Advantages of Pipesort for Data Cube Computation

✅ **Efficient Sorting –** Processes data incrementally instead of sorting everything at once.
✅ **Minimizes Memory Use –** Works with partial data rather than requiring full in-memory sorting.
✅ **Optimized for OLAP Queries –** Reduces computational cost for multidimensional aggregation.
✅ **Supports Streaming Data –** Can be applied to real-time data cube computations.

Conclusion

- **Data Cube Lattice** represents **all possible groupings** in a multidimensional data cube.
- **Pipesort** is an efficient **incremental sorting and aggregation** technique for computing **data cubes**.
- **Combining Data Cube Lattice with Pipesort** helps in **fast, memory-efficient OLAP operations**.

Business Intelligence

Business Intelligence is the analytical prozess, that transforms – fragmented – company and competition data in action-oriented knowledge about skills, positions, actions and targets in the regarded internal or external fiels of action (actors and processes).

[Grothe & Gensch 2000]

- Analytical process: planning, deciding and directing
- Omniscient data integration and provision
- Action-oriented knowledge: communication + information + knowledge representation

Part X

Business Intelligence Anwendungen

Business Intelligence Anwendungen

- Definition
- Use Cases
- Report & BSC

Definition

Business Intelligence

Diverse definitions:

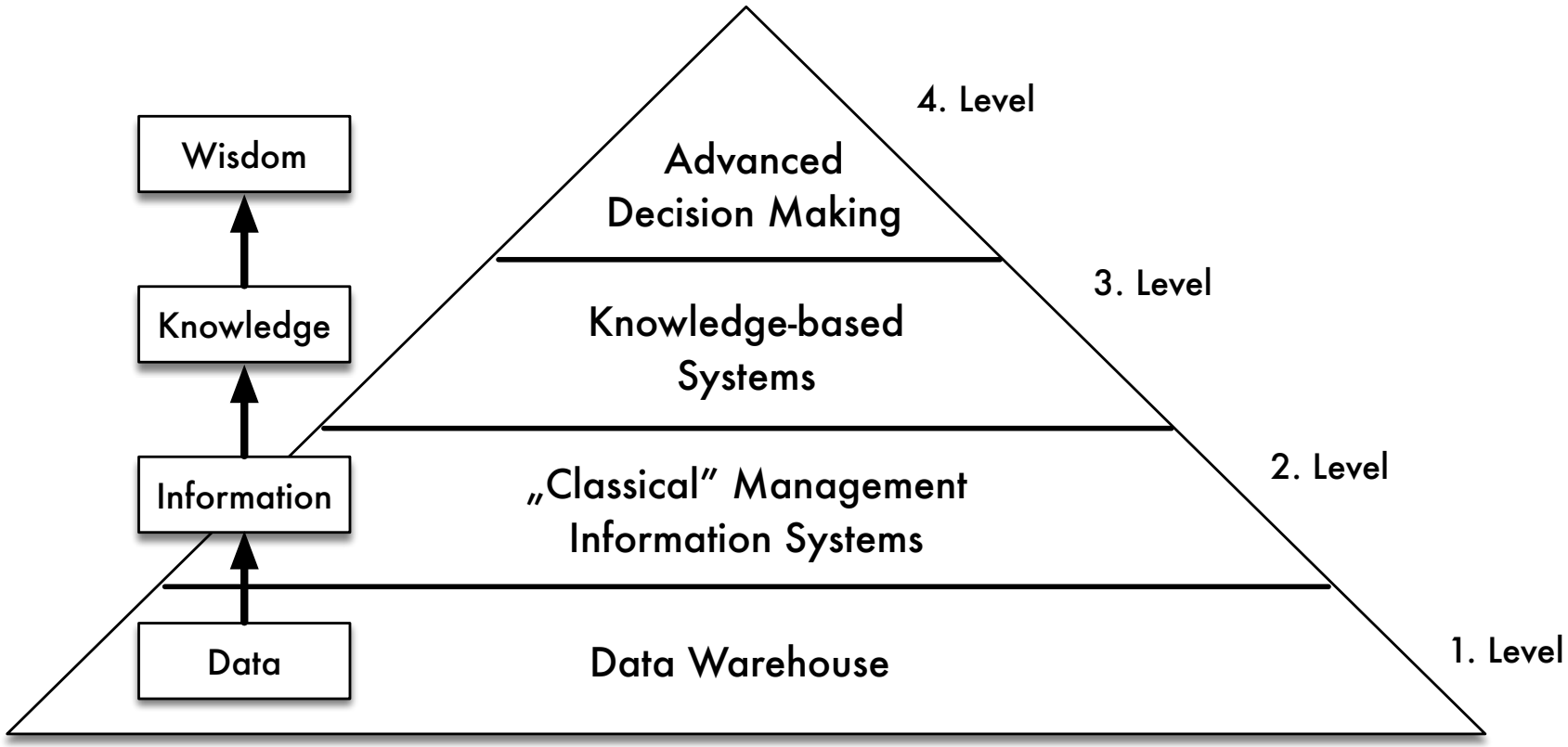
- 1989 term Business Intelligence coined [Dresner 1989]
- from the 60's (since data processing):
  - Management Information Systems
  - Management Support Systems
  - Executive Information Systems
- Differentiation:
  - In a narrower sense
  - Analysis-oriented
  - In a broader sense

Intelligence

Terminology:

- Finding orders,
- Rules for commonalities (consilience),
- Rules for co-occurence and sequential occurrences of events,
- Targeted collection and transfer of information,
- Information logic

Knowledge Pyramid



Business Intelligence

- Data- and information processing for the management
- Information logistics: filtering of information
- MIS: fast and flexible evaluations
- Early warnings in companies ("Alerting")
- BI = Data Warehousing
- Information and knowledge storage
- Prozess of gathering $\rightarrow$ Diagnosis $\rightarrow$ Therapy $\rightarrow$ Forecast $\rightarrow$ Control

[Mertens 2002]

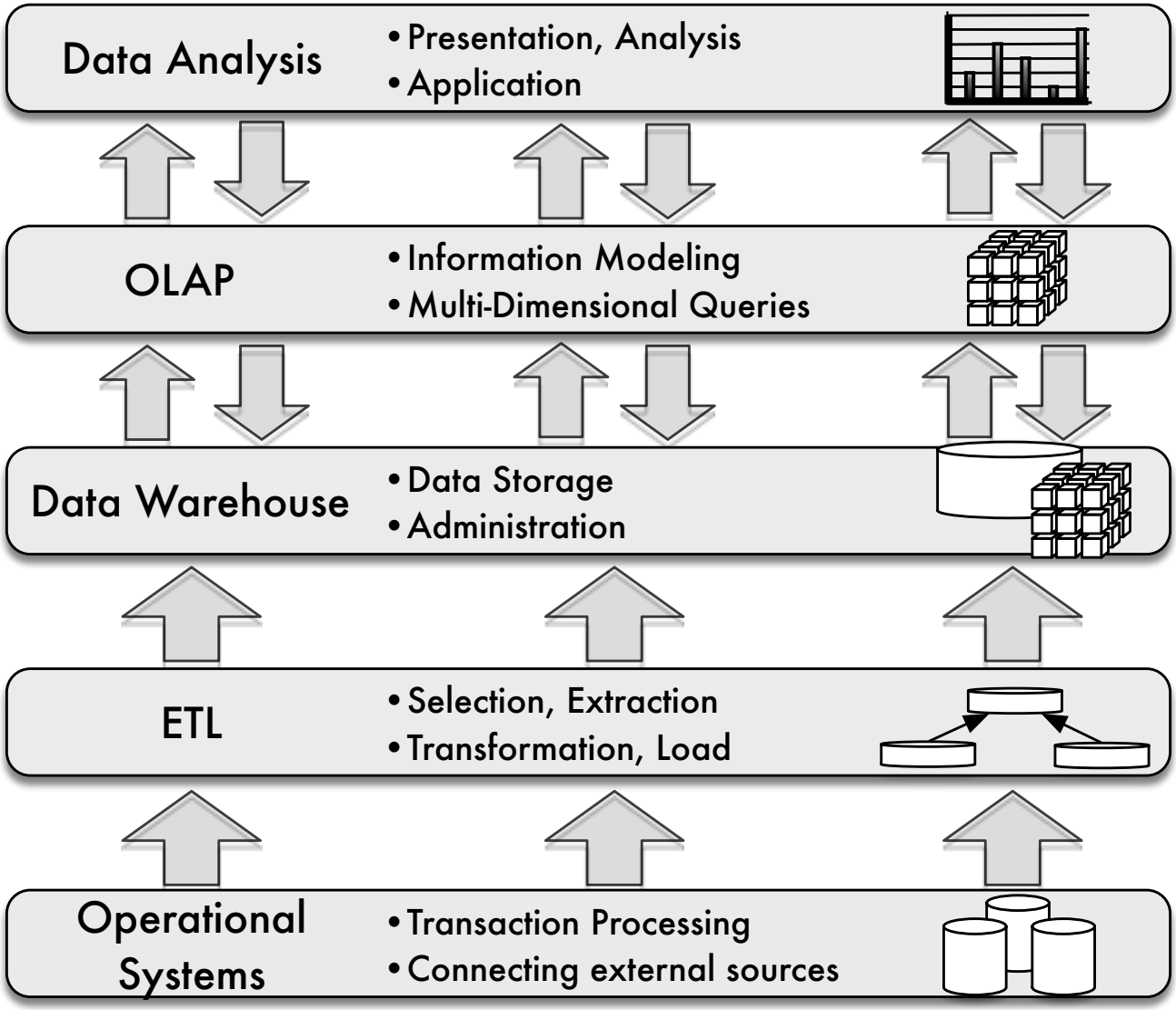
Business Intelligence

Business Intelligence is the analytical prozess, that transforms – fragmented – company and competition data in action-oriented knowledge about skills, positions, actions and targets in the regarded internal or external fiels of action (actors and processes).

[Grothe & Gensch 2000]

- Analytical process: planning, deciding and directing
- Omniscient data integration and provision
- Action-oriented knowledge: communication + information + knowledge representation

Business Intelligence Prozess



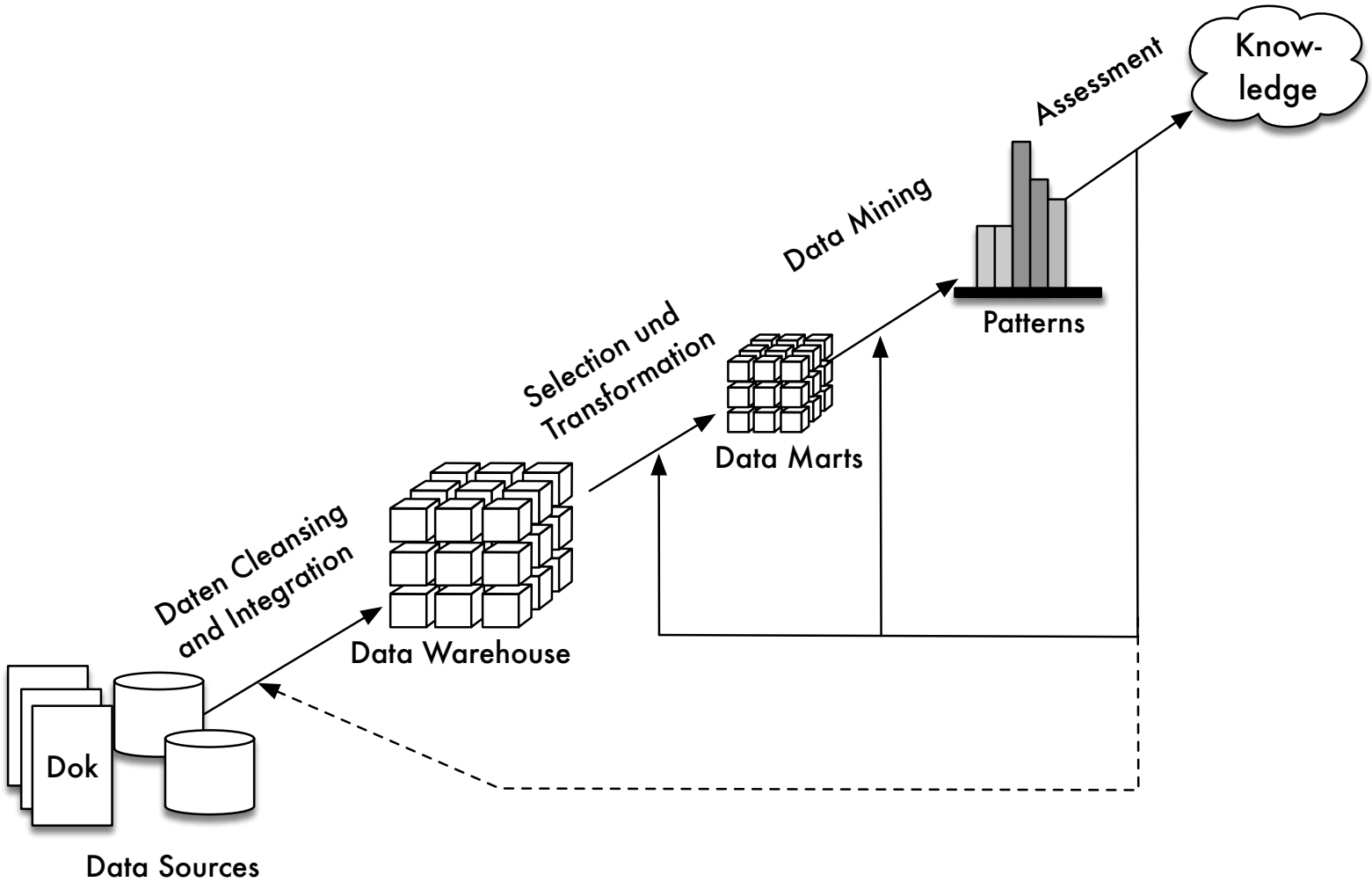
Data Warehouse and Business Intelligence

- Data Warehouse is a central information storage
- BI: methods to connect quantitative, qualitative, internal and external information
- DW data needs to be accordingly filtered and aggregated to represent personalized information / knowledge
- Data Mart is starting point for domain-specific analysis

Large data volume:

- Data in the OLAP area grows permanently
  $\hookrightarrow$ Overview of structure in the data by exploratory methods
- Data Mining pattern recognition

Knowledge Discovery Prozess



[Han & Kamber 2006]

Business Intelligence

Business Intelligence is the decision-oriented collection, preparation and presentation of business-relevant information.

[Schrödl 2006]

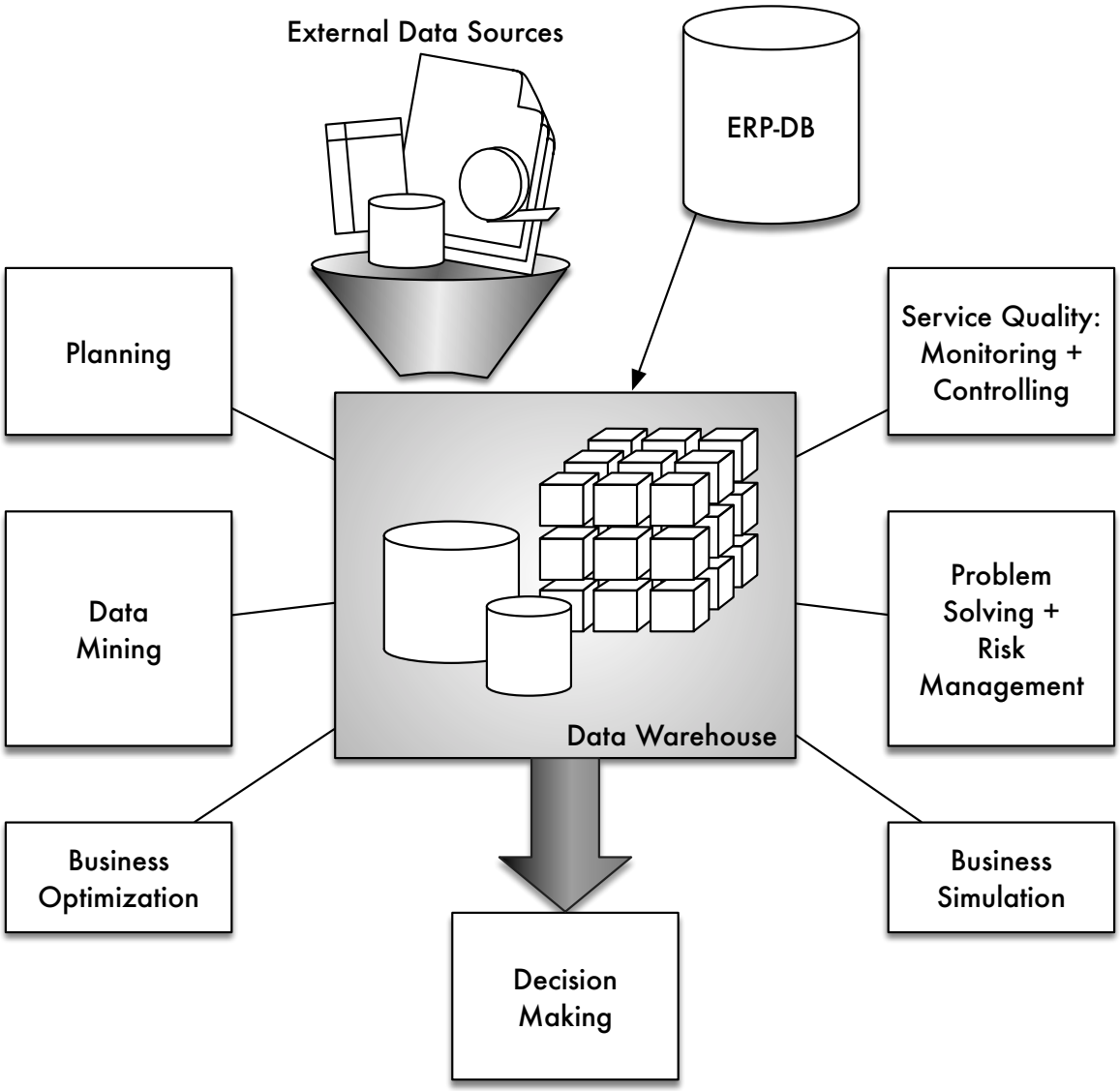
- Improve decision basis,
- Data collection: heterogeneous sources and requirements (e.g. security)
- Transform raw data in information (e.g, mathematical, rule based)
- Information representation for the user
- Concentrate on business relevance (optimization benefits & efforts)

BI Cycle

1. Quantification and qualification of business information
2. Analysis of the obtained data
3. Gaining insights supporting business processes
4. Evaluating the insights given the goals
5. Implementing the relevant insights in concrete actions

[Vitt et al. 2002]

Business Intelligence



Use Cases

Typical DW Use Cases

- Which clients do we have?
  ↪ Customer Relationship Management
- How do our costs develop?
  ↪ Supply Chain Management
- Where is further potential in our product range?
  ↪ Customer Basket Analysis
- ...

Typical Data Mining Methods

- Association rules – What has been bought together in a customer basket?
- Classification approaches – Which customer groups shall get special offers?
- Clustering – Which commonalities exist between our clients / suppliers?
- ...

Customer Basket Analysis

- Transactions at a coubter (transaction data base):
  - T1: {Müller-Thurgau, Riesling, Dornfelder}
  - T2: {Riesling, Erfurter Bock, Ilmenauer Pils, Anhaltinisch Flüssig}
  - T3: {Müller-Thurgau, Riesling, Erfurter Bock }
- Customer basket analysis: Which products are bought frequently together?
- Targets:
  - Optimization Shop Layout
  - Cross-Marketing
  - Add-On Sales

Association Rules

- Rule type:
  Body $\rightarrow$ Head [support, confidence]
- Example:
  - buys(X, "Red wine") $\rightarrow$ buys(X, "Erfurter Bock") [0.5%, 60%]
  - 98% of all clients buying Müller-Thurgau and Riesling pay by credit card.

Basic Definitions

according to [Agrawal und Srikant (1994)]

- Items $I = \{i_1, i_2, \ldots, i_m\}$ – Population of literals
- Itemset X: $X \subseteq I$
- Database D – Set of transactions $X \subseteq I$
- $X \subseteq T$
- Lexikographical sorting in T and X
- Length k of a itemset: number of elements
- k-Itemset: Itemset of length k

Basic Definitions (2)

- Support of the set X in D: share of transactions in D, that contain X:
  $supp(X) = \frac{|X|}{|D|}$
- Association rule: $A \rightarrow B$, with $A \subseteq I$, $B \subseteq I$ and $A \cap B = \emptyset$
- Support s of a association rule $A \rightarrow B$ in D: $s = supp(X \cup Y)$
- Confidence c of a association rule $A \rightarrow B$ in D: share of transactions, that contain B when they are present in A – $c = conf(B|A) = \frac{supp(A \cup B)}{supp(A)}$

Problem: Identify all association rules that in D exhibit a support $\geq$ minsup and a confidence $\geq$ minconf.

Example Association Rules

minsup = 20 %

TID	Items
1	Erfurter Bock, MT, Riesling
2	Erfurter Bock, MT, Dornfelder
3	Ilmenauer Pils, MT
4	Anhaltinisch Flüssig, Dornfelder, Riesling
5	Berliner Bräu, Dornfelder, Riesling
6	Kölnische Weisse, MT
7	Anhaltinisch Flüssig, Dornfelder

- $supp(MT) \approx 57\%$
- $supp(Riesling) = supp(Dornfelder) \approx 43\%$
- $supp(Erfurter\ Bock) = supp(Anhaltinisch\ Flüssig) \approx 29\%$
- $supp(Ilmenauer\ Pils) = supp(Berliner\ Bräu) = supp(Köln.\ Weisse) \approx 14\%$.
- potential candidates: MT, Riesling, Dornfelder, Erfurter Bock, Anhaltinisch Flüssig

Example Association Rules (2)

- possible combinations of all candidates:

Itemset	Support in %
(Erfurter Bock, MT)	$\approx$ 29
(Erfurter Bock, Riesling)	$\approx$ 14
(Erfurter Bock, Dornfelder)	$\approx$ 14
(Erfurter Bock, Anhaltinisch Flüssig)	0
(MT, Riesling)	$\approx$ 14
(MT, Dornfelder)	$\approx$ 14
(MT, Anhaltinisch Flüssig)	0
(Riesling, Dornfelder)	$\approx$ 29
(Riesling, Anhaltinisch Flüssig)	0
(Dornfelder, Anhaltinisch Flüssig)	$\approx$ 29

Apriori Algorithm

Input I, D, minsup
Output $\bigcup_k L_k$
C_k: candidates that shall be counted of length k
L_k: set of all frequent occurring itemsets of length k
initialize L_1:= 1-itemsets of I, k:= 2
WHILE $L_{k-1} \neq \emptyset$
    C_k := AprioriCandidateGeneration(L_{k-1});
    FOR EACH Transaction $T \in D$
        CT := Subset(C_k, T)
        // all candidates from C_k, that T contains
        FOR each candidate $c \in CT$ c.count++
    L_k := $\{c \in C_k | (c.count/|D|) \geq minsup\}$
    k++

Improving the Efficiency of the Apriori Algorithm

- Counting the support using a hash table
  - [Park, Chen, Yu 1995]
  - Hash tabele instead of a hash tree
  - k-itemset, whose bucket has a numerator smaller than the minimal support, cannot be frequent
    more efficient access to candidates, less accurate computation
- Transaction Reduction
  - [Agrawal & Srikant 1994]
  - Transactionens, that do not have a k-frequent itemset are redundant, i.e., they can be removed
  - Database scan is more efficient, but there is writing effort

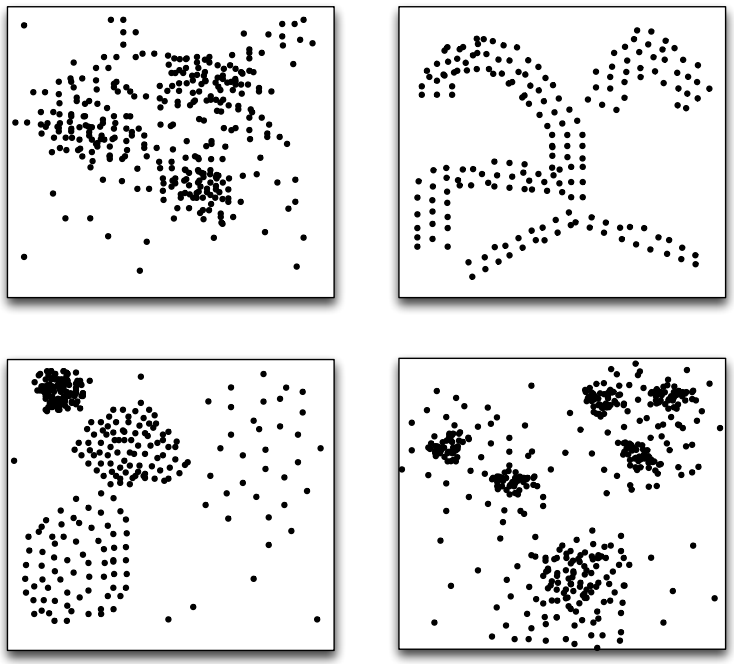
Improving the Efficiency of the Apriori Algorithm (2)

- Partitioning
  - [Savasere, Omiecinski & Navathe 1995]
  - Itemset only frequent when it is frequent in a partition
  - Exploiting the main memory (Partition)
  - Partition efficient, but effort for merging
- Sampling
  - [Toivonen 1996]
  - Application of Apriori on an excerpt (Sample)
  - Counting of found rules on the whole database

Cluster Approaches

- Identification of a finite set of groups in the data $\rightarrow$ Search for partitioning
- Similarity within a group
- Preferably significant difference between the groups

Occurring patterns (size, form, density):



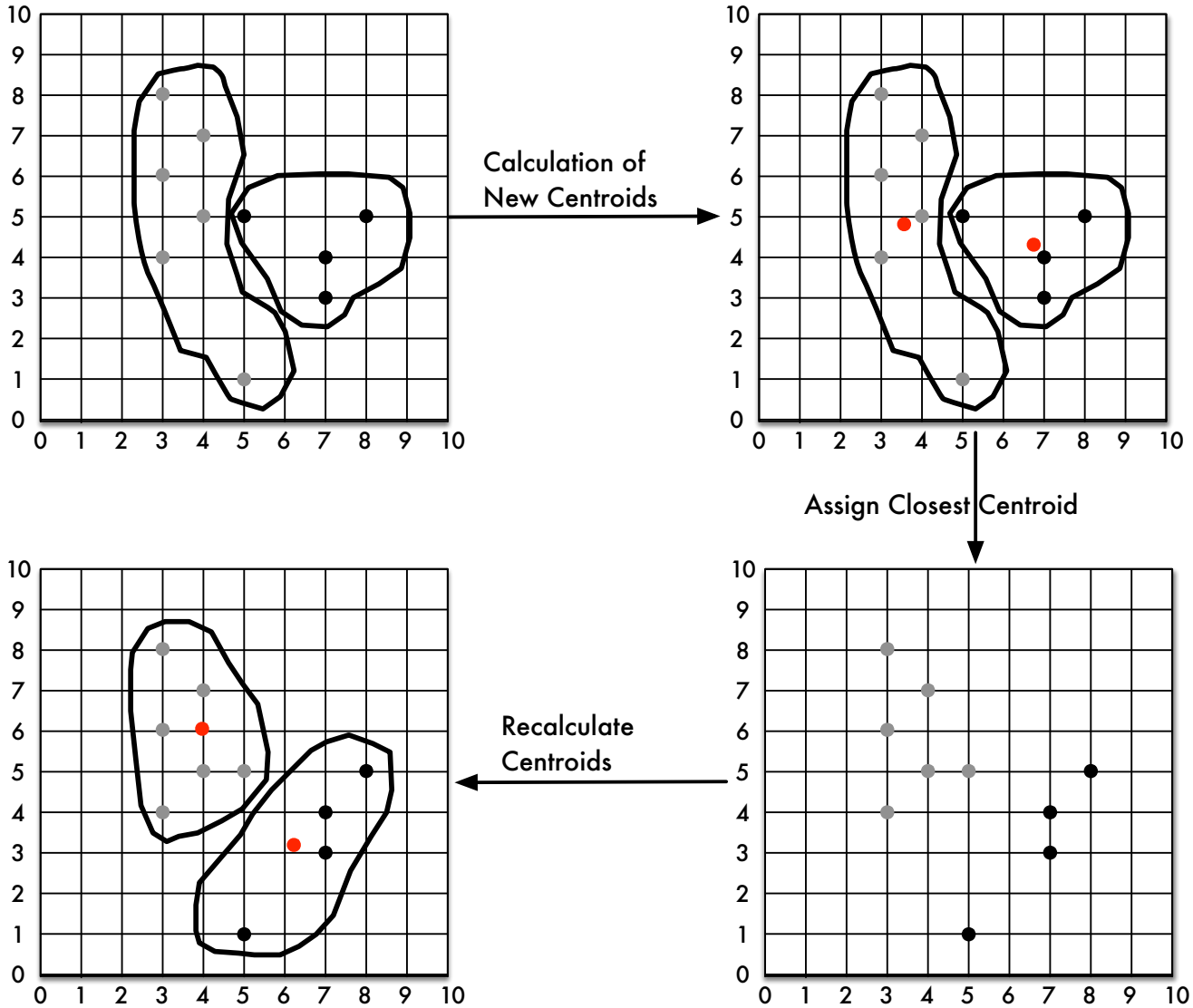
Distance functions

- Similarity metric $sim(objekt_1, objekt_2)$
- Distance function $dist(objekt_1, objekt_2)\ O \times O \rightarrow R_+$
  - small distance $\rightarrow$ similar, large distance $\rightarrow$ not similar
  - $dist(objekt_1, objekt_2) = 0$, given if $objekt_1 = objekt_2$
  - Symmetry: $dist(objekt_1, objekt_2) = dist(objekt_2, objekt_1)$
  - For metrics:
    $dist(objekt_1, objekt_3) \leq dist(objekt_1, objekt_2) + dist(objekt_2, objekt_3)$

Partitioning Clustering

Clustering through minimizing variance
Input: Tuple set D, numer of classes k
Output: Cluster C
Create an initial partitioning of D in k classes
Compute set $C^* = \{C_1, \ldots, C_k\}$ of
    centroids per class
$C := \{\}$
repeat
    $C := C^*$
    Partition: Create k classes by assigning
    each point to the closest centroid from C
    Compute centroids: Calculate the set
    $C^* = \{C_1^*, \ldots, C_k^*\}$ of centroids
    for the newly determined classes
until C = C^*

Cluster Approaches: Illustration



Advantages and Disadvantages

Advantages:

- linear effort per iteration, few iterations
- easy to implement
- k-means [MacQueen 1967]: most popular clustering algorithm

Disadvantages:

- Sensitive to noise and outliers
- Convex form of the clusters
- Fixed number of clusters
- Initial distribution important for runtime and end result

Classification: Example

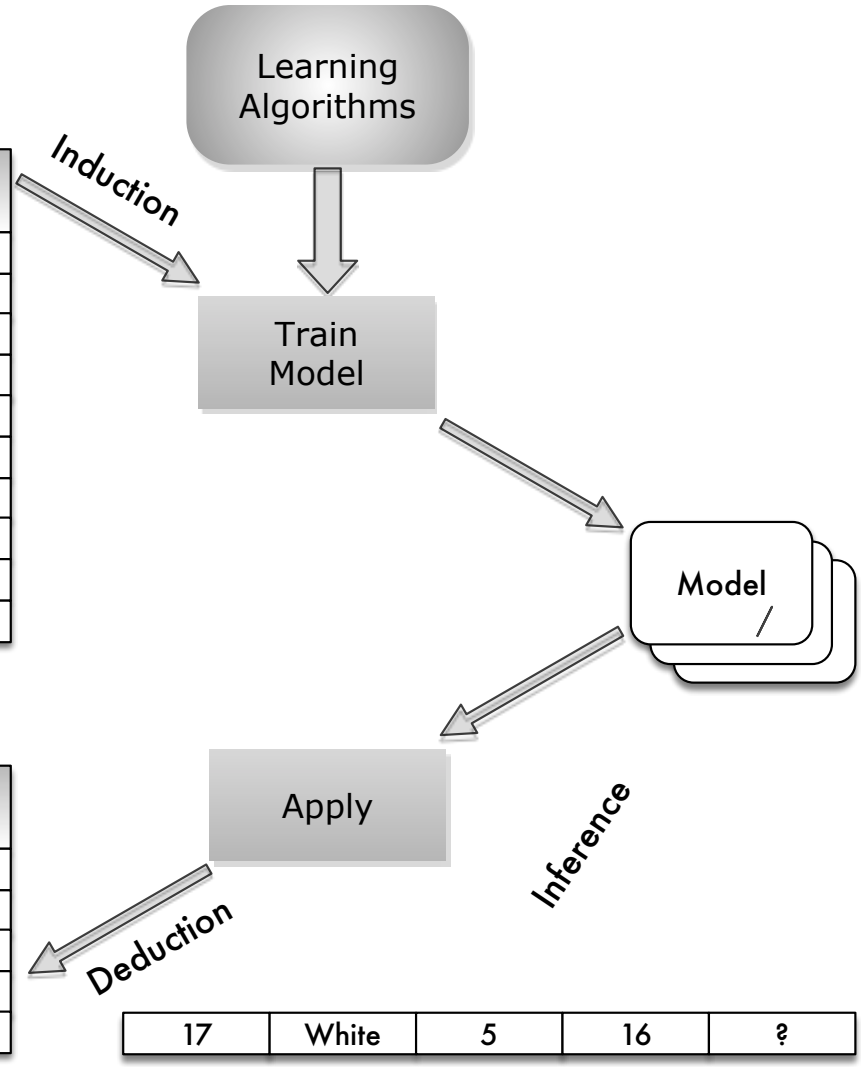
Do we like the Wine?

TID	Wine Color	Res. Sugar g/l	Alcohol	Class
1	White	18	10	Yes
2	Red	20	9	Yes
3	Rose	22	9	No
4	Rose	15	8	No
5	Red	30	5	Yes
6	White	18	10	Yes
7	Red	15	15	No
8	White	45	5	Yes
9	White	18	14	Yes
10	Red	8	10	No

Training Set

TID	Wine Color	Res. Sugar g/l	Alcohol	Class
11	Red	23	10	?
12	Rose	15	12	?
13	White	22	10	?
14	White	30	6	?
15	Red	12	14	?

Test Set

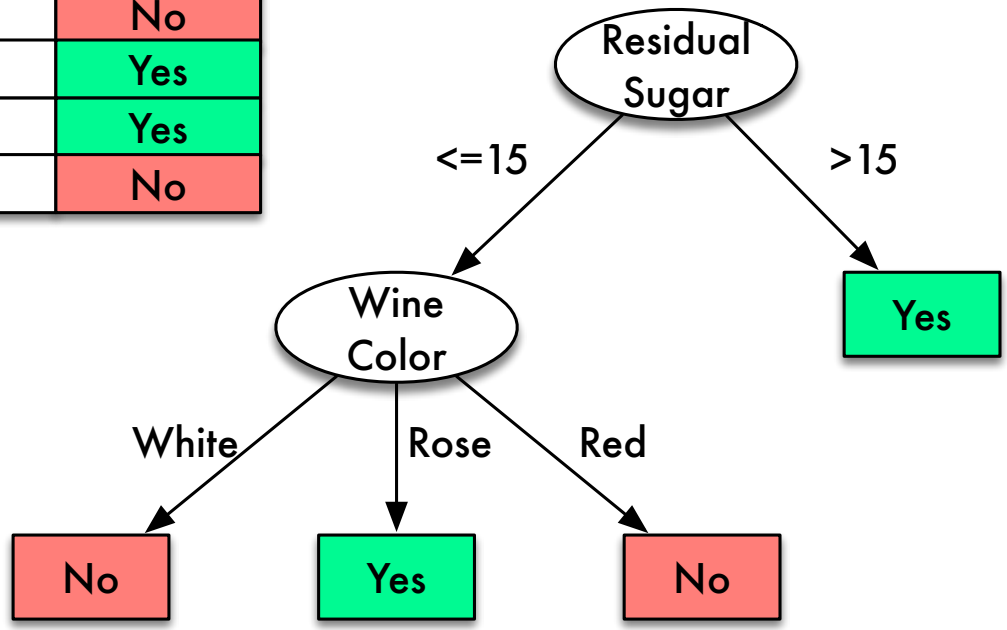


Classification

- Given is a set of object with attributes $o = (x_1, \ldots, x_d)$ and their membership to the set of classes C
- Search for classifier K for new objects $\rightarrow K : Objekts_{new} \rightarrow C$
- Class membership a-priori known $\rightarrow$ Difference to clustering approaches
- Similar to forecast (e.g., linear regression)

Classification Result

TID	Wine Color	Res. Sugar g/l	Alcohol	Yes/No
1	Red	23	12	Yes
2	White	15	10	No
3	Rose	14	10	Yes
4	White	30	6	Yes
5	Red	12	14	No



Classification Quality

		Forecast	
True labels		Member of class	Not member of class
	Member of class	True Positive	False Negative
	Not member of class	False Positive	True Negative

- Accuracy: $\frac{TP+TN}{TP+FN+FP+TN}$
- Precision: $p = \frac{TP}{TP+FP}$
- Recall: $r = \frac{TP}{TP+FN}$
- F-Measure: $F = \frac{2 \cdot TP}{2 \cdot TP+FN+FP}$

Classification Methods

- Decision Tree
- Rule-based
- Linear discriminant analysis by Fisher
- Categorical regression, Log-Linear models
- Neural networks
- Naive Bayes and Bayesian Belief Networks
- Support Vector Machines

Decision Tree

- Process: Splitting and Partitioning
- Explicites knowledge is found
- Easy to understand
- Easy to visualize

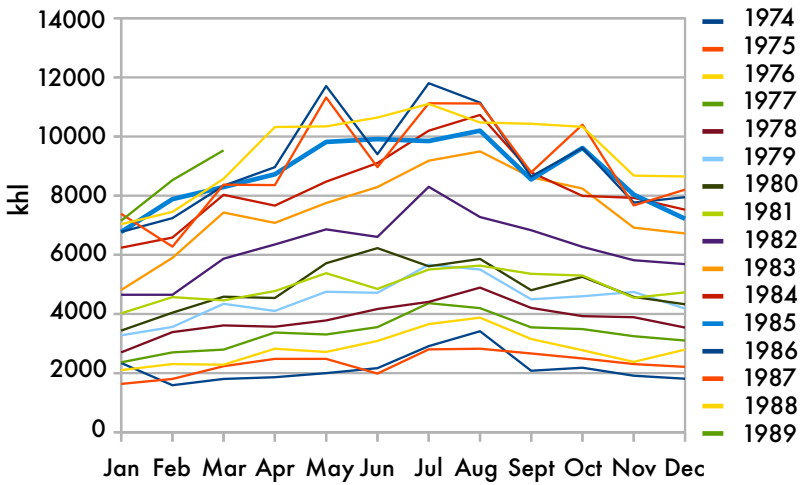
Algorithm for the Decision Tree

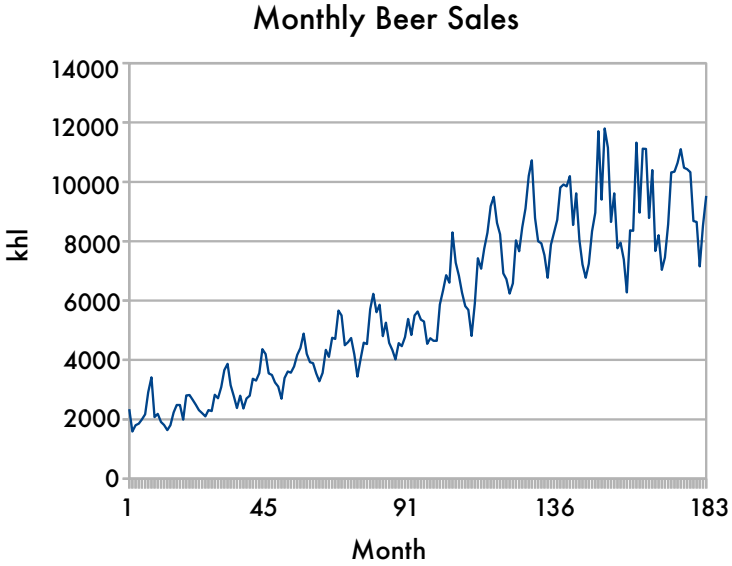
```
Input:  Training data
Initialization:  all data points (instances)
   belong to the root node
WHILE Split attribute exists OR data points
of a node in different classes
   Choose a split attribute (Splitting Strategy)
   Partition data points of a node
   according to the attribute
   Recursion for all partitions
```

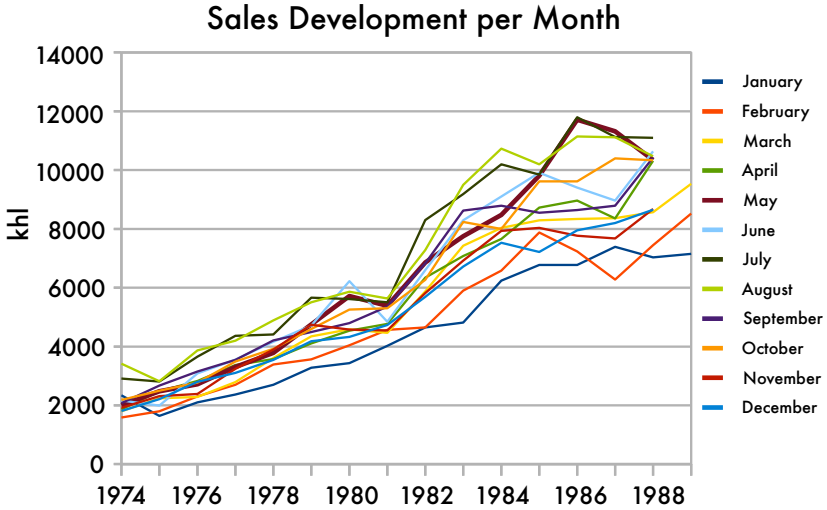
Forecast: Example

Monthly Beer Sales of a Brewery (khl)

1974	1975	1976	1977	1978	1979	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989
2339	1638	2101	2363	2697	3279	3438	4021	4646	4811	6236	6770	6771	7386	7034	7150
1588	1798	2307	2700	3388	3561	4044	4570	4646	5896	6582	7881	7237	6279	7449	8525
1800	2235	2281	2794	3609	4343	4584	4461	5868	7426	8029	8290	8335	8370	8569	9530
1858	2481	2827	3371	3570	4103	4536	4771	6346	7076	7661	8720	8966	8356	10320	
2001	2479	2713	3303	3783	4749	5711	5383	6857	7749	8471	9813	11709	11318	10340	
2169	1988	3083	3555	4163	4711	6225	4843	6602	8293	9103	9913	9402	8964	10641	
2911	2804	3657	4364	4405	5661	5609	5504	8295	9183	10198	9847	11799	11119	11100	
3414	2820	3872	4198	4890	5503	5860	5633	7278	9496	10725	10196	11147	11113	10474	
2077	2666	3149	3547	4206	4494	4800	5360	6829	8620	8785	8546	8645	8783	10427	
2184	2494	2773	3491	3923	4595	5256	5297	6269	8237	7994	9613	9615	10397	10329	
1913	2308	2382	3246	3893	4740	4576	4546	5814	6919	7929	8038	7765	7672	8677	
1809	2212	2798	3102	3543	4179	4330	4733	5686	6721	7527	7217	7948	8202	8651	

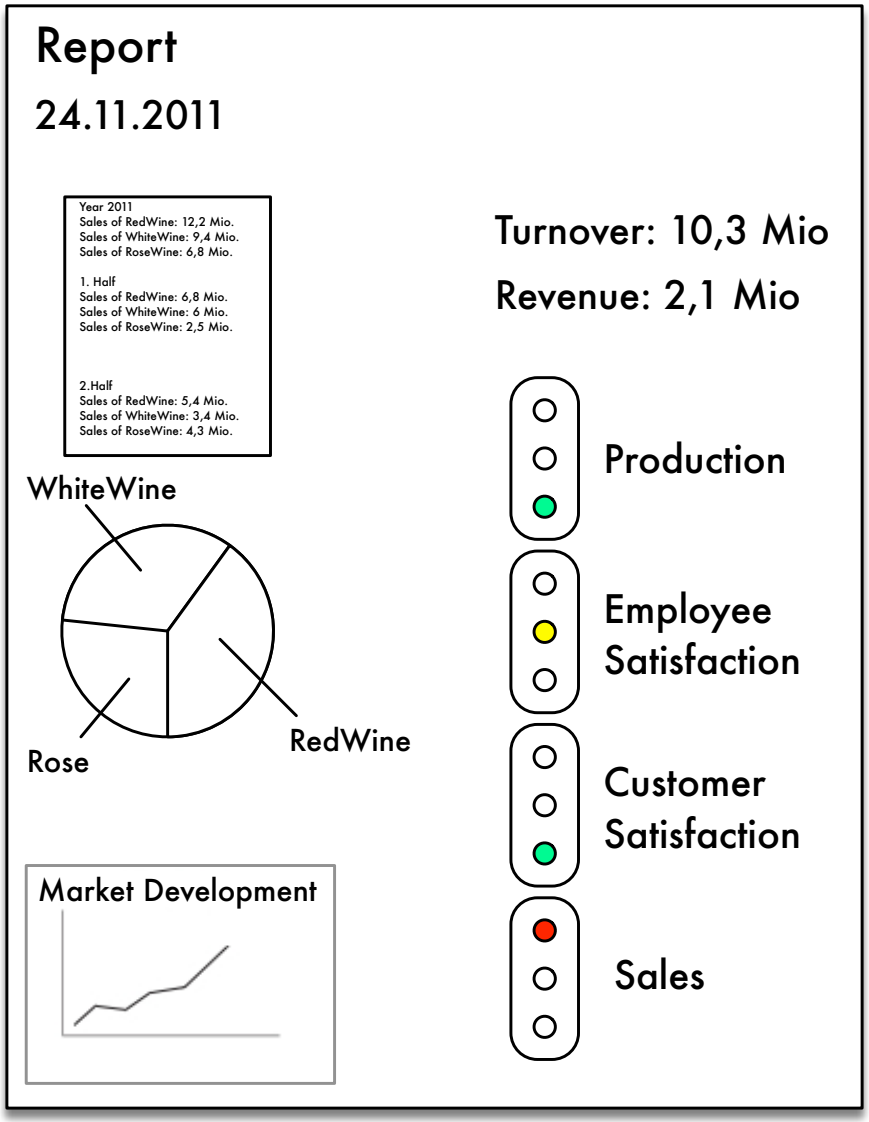




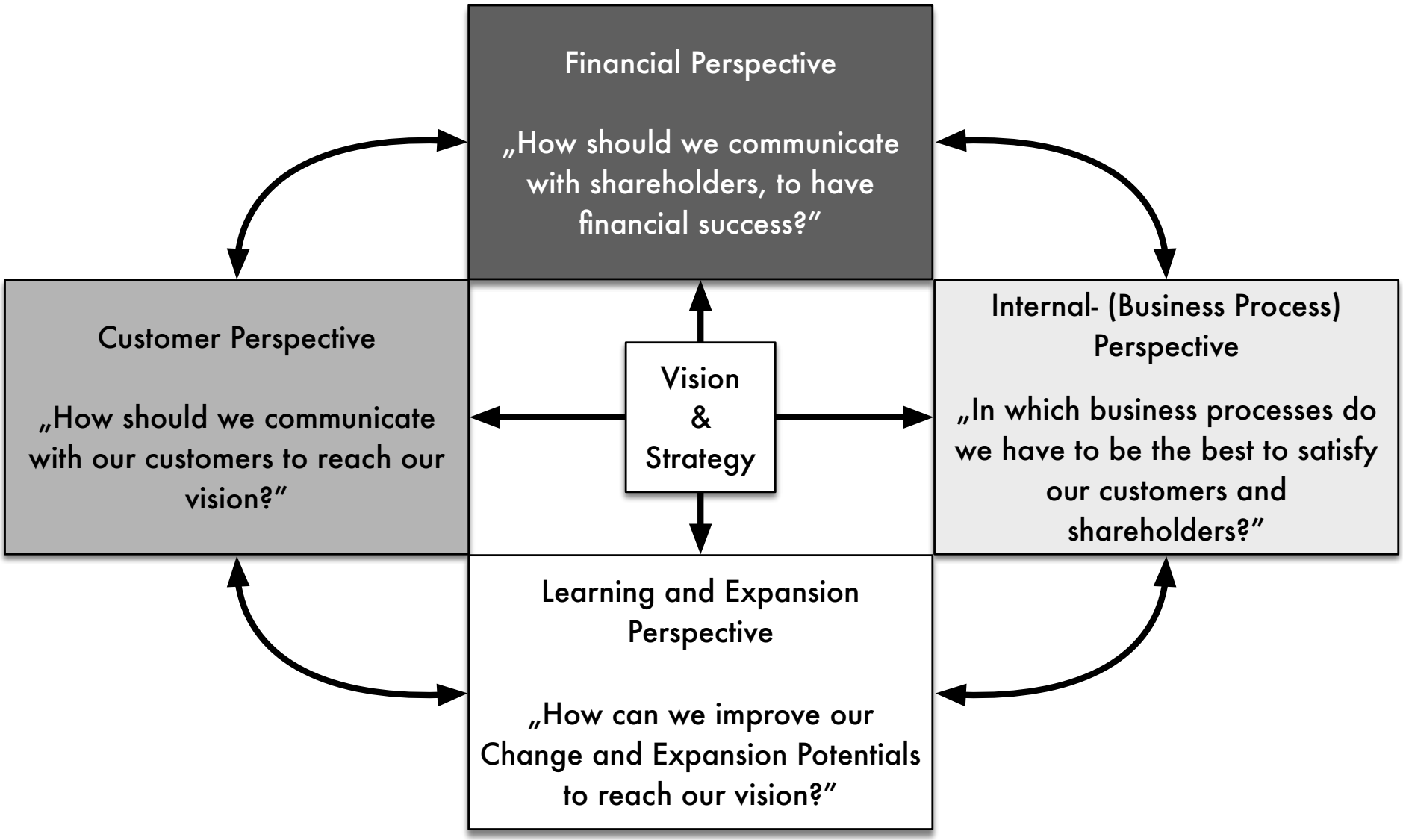


Report & BSC

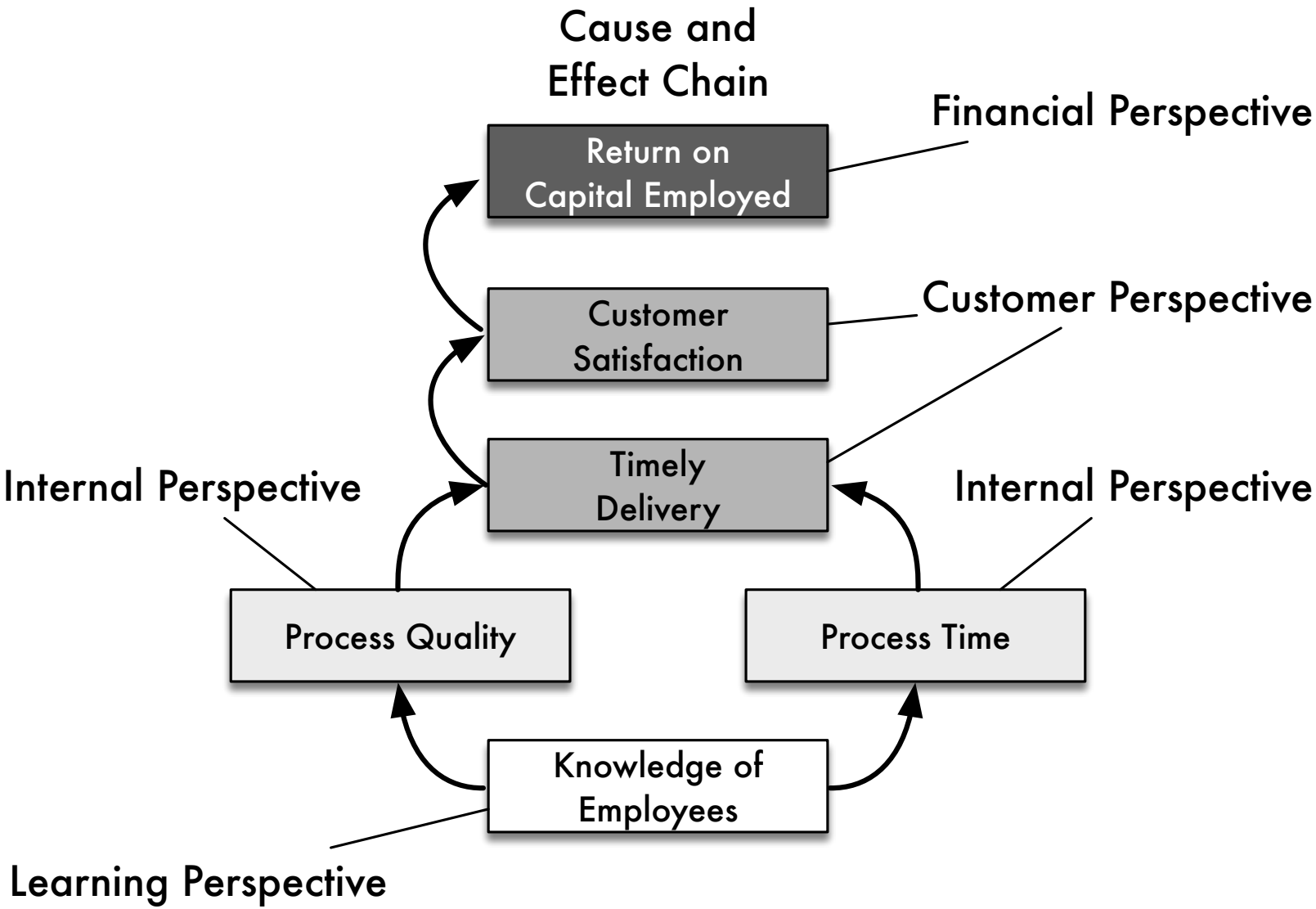
Reporting


Report
24.11.2011
Year 2011
Sales of RedWine: 12,2 Mio.
Sales of WhiteWine: 9,4 Mio.
Sales of RoseWine: 6,8 Mio.
1. Half
Sales of RedWine: 6,8 Mio.
Sales of WhiteWine: 6 Mio.
Sales of RoseWine: 2,5 Mio.
2.Half
Sales of RedWine: 5,4 Mio.
Sales of WhiteWine: 3,4 Mio.
Sales of RoseWine: 4,3 Mio.
WhiteWine
RedWine
Rose
Market Development
Turnover: 10,3 Mio
Revenue: 2,1 Mio
Production
Employee Satisfaction
Customer Satisfaction
Sales

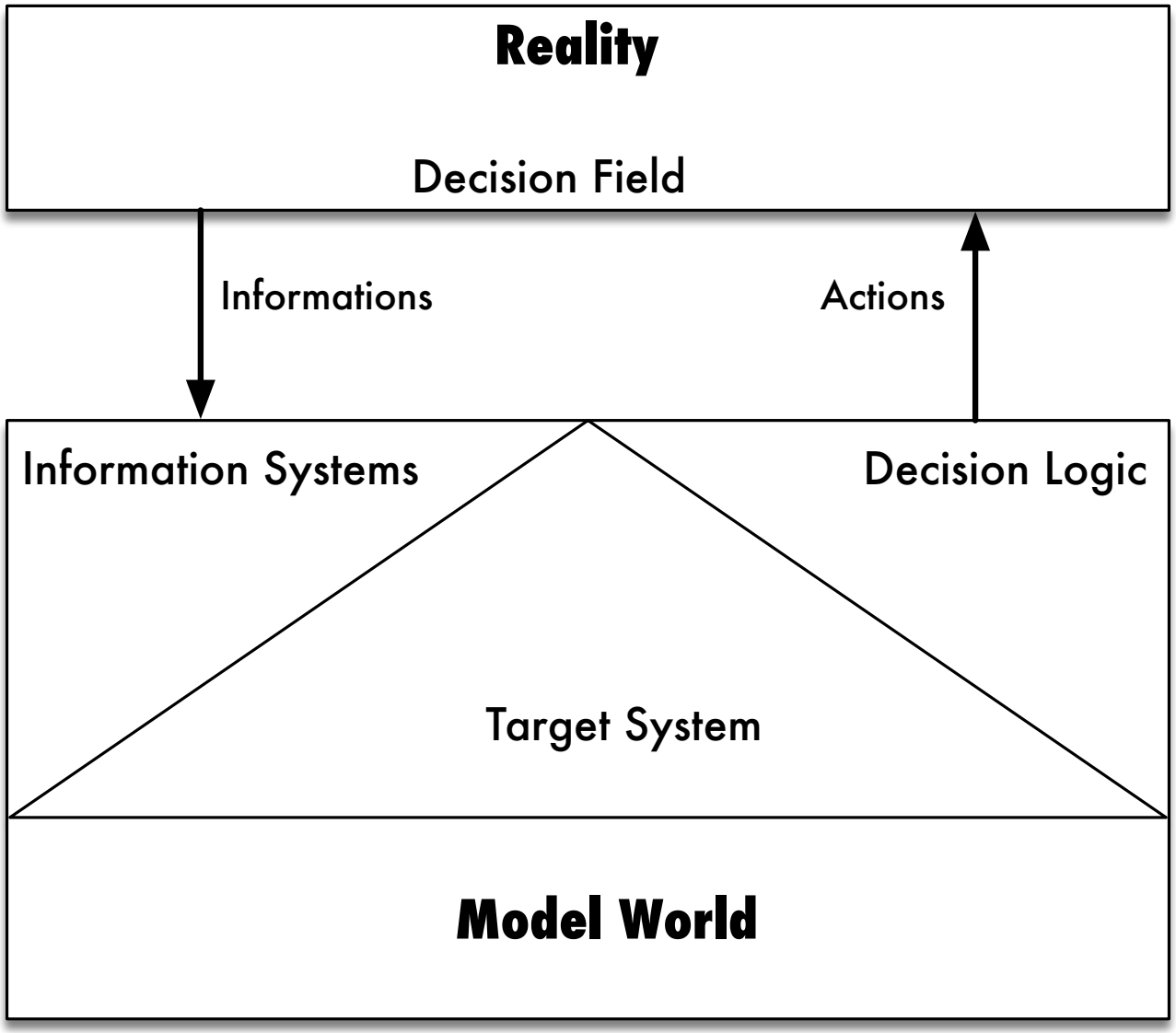
Balanced Scorecard



Interdependence



Decision Support



K-Means Clustering (Unsupervised Learning)

🔹 What is K-Means Clustering?

K-Means is a **machine learning algorithm** used for **clustering** data points into **groups (clusters)** based on their similarities. It is an **unsupervised learning algorithm**, meaning it doesn't need labeled data.

🔹 How It Works (Step-by-Step)

1. **Choose the Number of Clusters (K)**
   - You decide how many clusters (K) you want to create in your data.
2. **Initialize K Random Centroids**
   - Pick **K random points** as the initial "centroids" (center points of clusters).
3. **Assign Data Points to the Nearest Centroid**
   - Each data point is assigned to the **closest centroid**, forming clusters.
4. **Update the Centroids**
   - For each cluster, calculate the **average (mean)** of all points in that cluster and move the centroid to this new mean position.
5. **Repeat Steps 3 and 4**
   - Keep **reassigning points** and **updating centroids** until they **stop moving** or reach a **maximum number of iterations**.

🔹 Example of K-Means Clustering

Imagine you have a store, and you want to group your **customers** into clusters based on their **spending habits**.

Customer	Monthly Spending ($)	Number of Purchases
Alice	500	20
Bob	1000	50
Charlie	450	15
David	1200	55

If K = 2 (Two Clusters):

- One group might contain **low-spenders** (Alice, Charlie).
- The other group might contain **high-spenders** (Bob, David).

Visualization of Clusters:

```java
Cluster 1: Low Spending (Alice, Charlie)
Cluster 2: High Spending (Bob, David)
```

🔹 When to Use K-Means?

✅ When you want to **group similar data points** together.
✅ When you **don't know the labels** in advance.
✅ For **customer segmentation, image compression, anomaly detection, etc.**

2️⃣ Association Rule Mining (Finding Patterns in Data)

🔹 What is Association Rule Mining?

Association Rule Mining is a technique used to **find relationships (patterns) between items** in large datasets. It is commonly used in **market basket analysis** to find which items are frequently bought together.

🔹 How It Works (Step-by-Step)

1. **Collect Transaction Data**
   - Example: A supermarket records purchases of different items by customers.
2. **Find Frequent Itemsets**
   - Identify which **items appear together** most frequently.
3. **Generate Association Rules**
   - Find rules like "**If a customer buys A, they are likely to buy B**".
   - Example: **{Milk, Bread} → {Butter}** (People who buy milk and bread also buy butter).

🔹 Example of Association Rule Mining

Imagine you own a grocery store and analyze customer transactions:

Transaction ID	Items Purchased
1	Milk, Bread, Butter
2	Milk, Bread
3	Milk, Butter
4	Bread, Butter

From this data, you might discover the following **association rules**:

- **Rule 1**: If a customer buys **Milk & Bread**, they also buy **Butter** (**Confidence: 80%**).
- **Rule 2**: If a customer buys **Bread**, they are likely to buy **Milk** (**Confidence: 75%**).

🔹 Key Metrics in Association Rule Mining

- **Support**: How often an itemset appears in transactions.
- **Confidence**: How often the rule is true (i.e., if someone buys X, how often do they buy Y?).
- **Lift**: How much more likely Y is purchased when X is bought, compared to if there was no relationship.

🔹 When to Use Association Rule Mining?

✅ To discover **hidden patterns** in data.
✅ To make **recommendations** (Amazon, Netflix, grocery stores, etc.).
✅ For **fraud detection**, **medical diagnosis**, and **website analysis**.

3️⃣ Decision Trees (Supervised Learning, Classification & Prediction)

🔹 What is a Decision Tree?

A **Decision Tree** is a flowchart-like structure used for **classification** and **prediction**. It makes decisions by **splitting data** based on questions or conditions.

🔹 How It Works (Step-by-Step)

1. **Start with a Root Node (Question)**
   - Example: "Is the customer's salary > $50,000?"
2. **Split Data Based on Answers**
   - If **YES**, follow one branch.
   - If **NO**, follow another branch.
3. **Keep Splitting Until a Final Decision (Leaf Node) is Reached**
   - Example: "Will the customer buy a luxury car?"

🔹 Example of a Decision Tree for Loan Approval

Imagine a bank needs to decide **whether to approve a loan application**.

Customer	Income ($)	Credit Score	Loan Approved?
Alice	40,000	700	No
Bob	80,000	750	Yes
Charlie	35,000	600	No
David	90,000	800	Yes

Decision Tree:

yaml Copy

```
          Is Income > 50K?
            /       \
          Yes        No
        /     \       \
Credit Score>700?   Deny Loan
    /       \
```

```
   Yes       No
Approve   Deny Loan
```

- If **income > $50,000** and **credit score > 700**, **loan is approved**.
- If **income > $50,000** but **credit score < 700**, **loan is denied**.
- If **income < $50,000**, **loan is denied**.

🔹 Advantages of Decision Trees

✅ **Easy to Understand** (Like a flowchart).
✅ **Works with both numerical & categorical data**.
✅ **No need for complex feature scaling**.

🔹 When to Use Decision Trees?

✅ For **classification problems** (e.g., spam detection, loan approval, medical diagnosis).
✅ When data can be easily divided into **yes/no** conditions.
✅ When interpretability is important.

🔹 Summary Table

Technique	Type	Used For	Example
K-Means Clustering	Unsupervised	Grouping similar data points	Customer segmentation (low/high spenders)
Association Rule Mining	Pattern Mining	Finding item relationships	Market Basket Analysis (Milk → Bread)
Decision Trees	Supervised	Classification & Prediction	Loan approval, spam detection