Advance Data Management

Chapter-1: Introduction and Challenges

 # Database properties

1. Data management
2. Scalability
3. Heterogeneity
4. Efficiency
5. Persistence
6. Reliability
7. Consistency
8. Non-redundancy
9. Multi-User support

→ Data management

CRUD Operations:

1. Create (C): Adding new data to the database.

- Example (SQL):

```
INSERT INTO student VALUES (4711, 'Liesl Weppen', 'MOBI-ADM');
```

2. Read (R): Retrieving data from the database.

- Example (SQL - Read by ID):

```
SELECT * FROM student WHERE student_id = 4711;
```

- Example (SQL - Read by Query):

```
SELECT name FROM student WHERE course = 'MOBI-ADM';
```

3. Update (U): Modifying existing data in the database.

- Example (SQL):

```
UPDATE student SET course = 'New Course' WHERE student_id = 4711;
```

4. Delete (D): Removing data from the database.

- Example (SQL):

```
DELETE FROM student WHERE student_id = 4711;
```

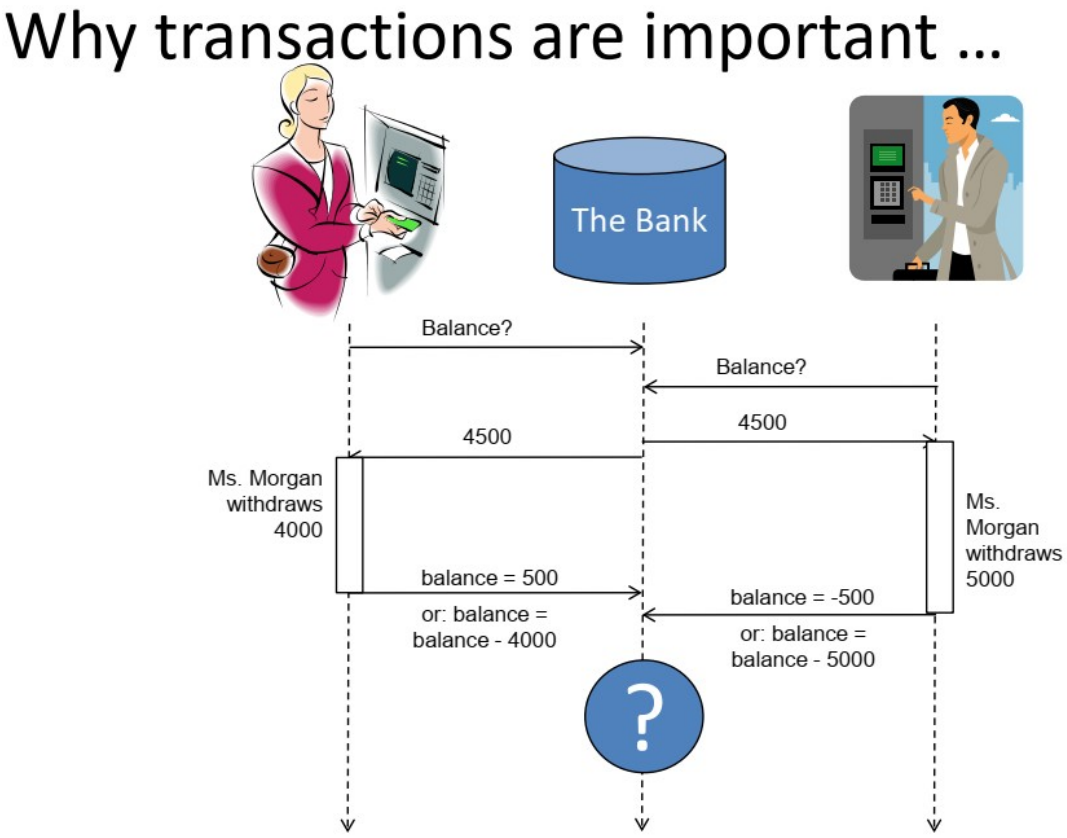
Transactions ensure a group of database operations are treated as a single unit of work.

- Example (Pseudocode):

```
BEGIN TRANSACTION;
```

```
READ account_balance FROM account WHERE account_id = 'sender';
IF account_balance >= amount THEN
    UPDATE account SET account_balance = account_balance - amount WHERE
account_id = 'sender';
    UPDATE account SET account_balance = account_balance + amount WHERE
account_id = 'receiver';
    COMMIT;  // All operations succeed, commit changes
ELSE
    ROLLBACK;  // Roll back changes if any operation fails
END IF;
```

In the transaction example, if all operations (reading sender's account balance, updating sender's account balance, and updating receiver's account balance) succeed, the changes are committed. Otherwise, if any operation fails (e.g., insufficient funds), all changes are rolled back to maintain data integrity.



1. **Scalability:**
    - Scalability refers to the ability of a database system to handle increasing amounts of data or workload without sacrificing performance.
    - Example: A NoSQL database like MongoDB is designed to scale horizontally by adding more servers to distribute the load as data volume increases.
2. **Heterogeneity:**

- Heterogeneity refers to the ability of a database system to handle different types of data (e.g., structured, semi-structured, unstructured) and integrate with diverse data sources.
- Example: A data warehouse that integrates data from various sources such as relational databases, spreadsheets, and web services.

3. Efficiency:

- Efficiency refers to the ability of a database system to perform operations quickly and with minimal resource utilization.
- Example: Indexing frequently queried columns in a database table to speed up data retrieval operations.

4. Persistence:

- Persistence refers to the durability of data, ensuring that it remains intact and available even after system failures or shutdowns.
- Example: Writing data to disk in a database system to ensure that it persists even if the system crashes.

5. Reliability:

- Reliability refers to the trustworthiness and consistency of data stored in a database, ensuring that it is accurate and can be relied upon for decision-making.
- Example: Using transactions in a database system to ensure the atomicity, consistency, isolation, and durability (ACID) of operations.

6. Consistency:

- Consistency ensures that data remains in a valid state during and after transactions, enforcing integrity constraints and preventing data corruption.
- Example: A banking application deducts funds from one account and adds them to another in a single transaction to maintain consistency.

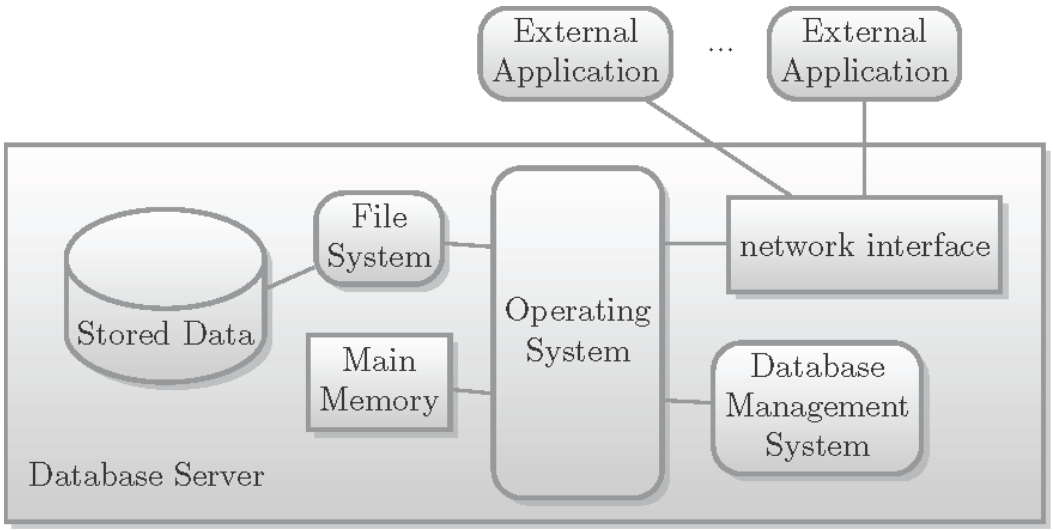
7. Non-redundancy:

- Non-redundancy refers to the practice of minimizing data duplication within a database to conserve storage space and reduce the risk of inconsistencies.
- Example: Normalizing database tables to eliminate redundant data and maintain data integrity.

8. Multi-User Support:

- Multi-user support allows multiple users to access and manipulate data concurrently in a database system while ensuring data integrity and consistency.
- Example: A web-based application allows multiple users to log in, view, and update their profiles simultaneously without conflicts.

 # Database Components: Database management system (DBMS): software component is charge of all DB operations – Data is stored on the disk or in main memory, organized in Pages (chunks of equal size) – DBMS often rely on OS to access File system and Main memory – Applications interact by sending data.



 # Query Processing:

- **Goal:** Quickly retrieve data in response to user queries.
- **Page Buffer:** Special area in main memory managed by DBMS, acting as a cache for recently accessed data pages.
- **Page Hit:** Query results are already in the page buffer, enabling fast data retrieval.
- **Page Miss:** Query results need to be loaded from disk into the page buffer, potentially slowing down data retrieval due to disk I/O.

 # What happens if page miss?

when a page miss occurs, the DBMS retrieves the required page from disk, processes the relevant data within the page buffer, and eventually writes the modified page back to disk, balancing performance with data safety considerations.

 # Database Design:

Database design is a crucial phase that occurs before implementing a Database Management System (DBMS).

In essence, the database design phase lays the foundation for the efficient and effective organization of data within the database system, ensuring that it meets the requirements and expectations of its users and applications.

→ Design criteria

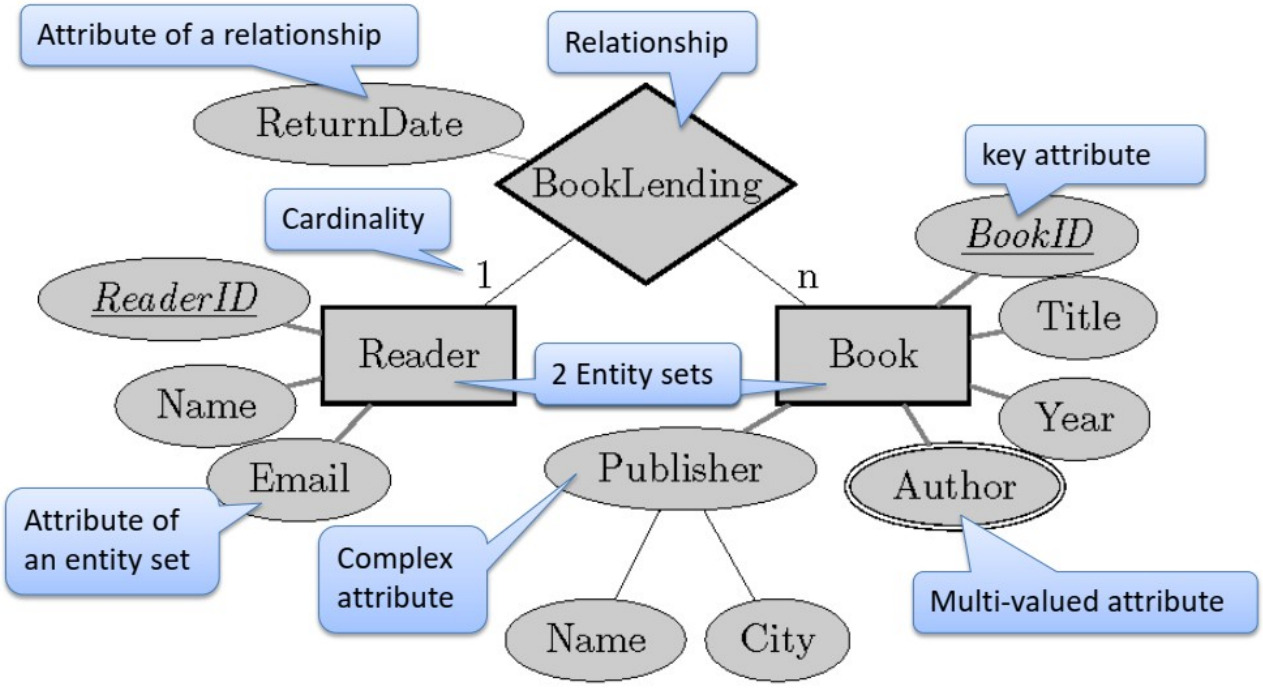
1. Completeness
– All aspects of the information needed by the applications should be covered
2. Soundness
– All information aspects and relationships are modeled correctly
3. Minimality
– No unnecessary or logically redundant information should be modeled
– (however, sometimes redundancy enhances performance)
4. Readability
– No complex encoding should be used to describe the information

– Entity and attribute names should be self-explanatory
5. Modifiability
– Changes to the data structure should be possible
6. Modularity
– The data model should be devided into subsets that form logically coherent entities
– Also helps with modifiability

 # In database design, several languages are used for modeling and representing the structure and relationships of data. Two popular examples are:

1. ER (Entity-Relationship) Model:

- **Description:** ER Model has a long history in conceptual modeling.
- **Functionality:**
  - Models entity sets (groups of entities with similar attributes) along with their attributes and relationships.
  - Supports extensions to represent concepts like generalization/specialization (similar to inheritance in object-oriented programming).
- **Use Case:** Often used for conceptual modeling in the early stages of database design.



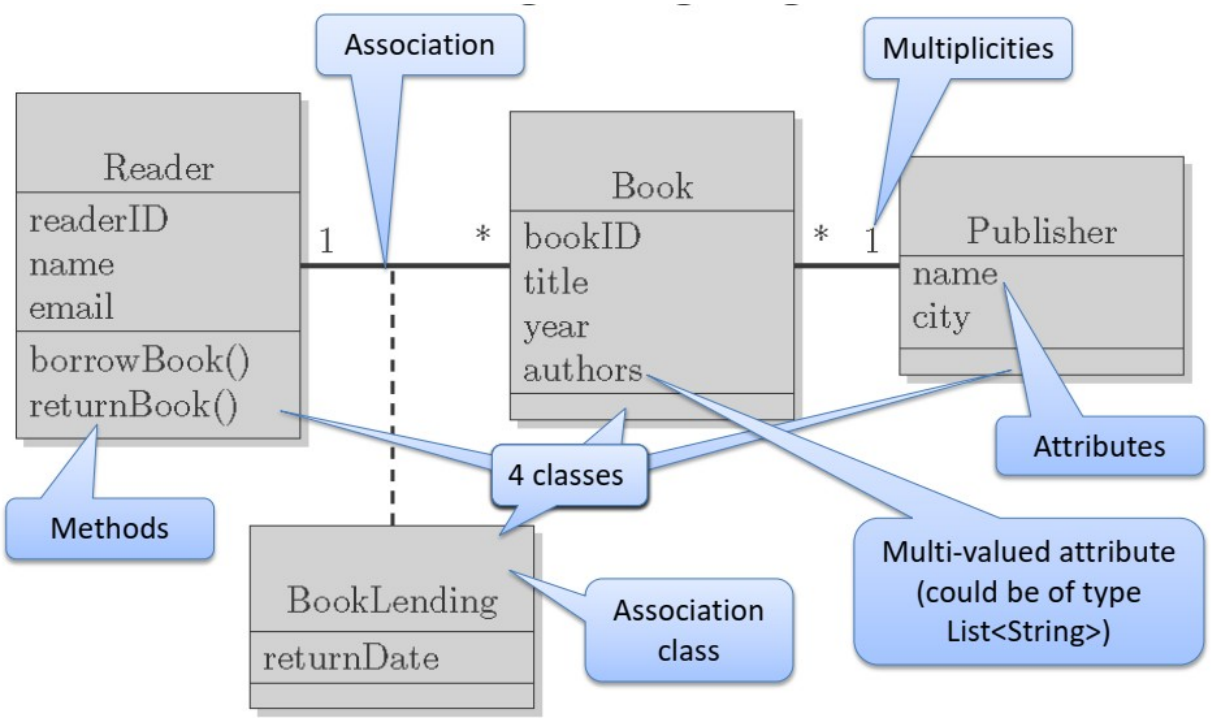
ER Model

1. UML (Unified Modeling Language):

- **Description:** UML is a standard defined by the Object Management Group (OMG).
- **Functionality:**
  - Can model not only entity sets (equivalent to classes in object-oriented programming) but also methods, objects, activities, and interactions.

- Provides a comprehensive modeling framework for software systems, including databases.

- **Use Case:** Widely adopted for modeling various aspects of software systems, including database design, due to its versatility and standardization.



Unified Modeling Language

The Relational Model

 # Relational schema: A relational schema is a relational database consists of a set of tables. The structure of tables is defined in the schema.

Constraint is a rule or condition enforced on data to maintain integrity and consistency.

- Types of constraints include:

1. **Primary Key:** Ensures uniqueness of records in a table.
2. **Foreign Key:** Establishes relationships between tables.
3. **Unique:** Ensures uniqueness of values within a column or set of columns.
4. **Check:** Verifies data against a specified condition.
5. **Not Null:** Requires that a column must have a value (not null).

Referential integrity is a crucial concept in relational database management systems (RDBMS) that ensures the consistency and integrity of relationships between tables. Here's how it works:

- **Foreign Key Constraints:** These constraints define relationships between tables. They ensure that values in a column (foreign key) in one table correspond to values in another table (primary key).

- **Example:** Consider two tables, "Book" and "BookLending." The "BookLending" table has a foreign key referencing the primary key of the "Book" table.

ACID Properties:

- **Atomicity:**
    - Transactions are "all or nothing."
    - If any part of a transaction fails, the entire transaction is rolled back.
- **Consistency:**
    - Before and after a transaction, the database remains consistent.
    - Constraints are always fulfilled, ensuring data integrity.
- **Isolation:**
    - Transactions are executed as if they are alone on the database.
    - Ensures that concurrent transactions do not interfere with each other.
- **Durability:**
    - Ensures that committed data persists even after system failures.
    - Modified data from successful transactions is durable and survives failures.

 # ER2RM:

The five main steps of the ER2RM (Entity-Relationship to Relational Model) conversion process are:

1. **Entities:** Create tables for each entity, including single-valued attributes.
2. **One-to-One Relationships:** Extend one side with a foreign key referencing the other side.
3. **One-to-Many Relationships:** Extend the "many" side with a foreign key referencing the "one" side.
4. **Many-to-Many Relationships:** Create a new table with foreign keys from both entities involved.
5. **Multivalued Attributes:** Create a new table for each multivalued attribute, linking it to the entity's primary key.

These steps ensure a systematic conversion from an ER diagram to a relational database schema while preserving the structure and relationships of the original model.

 # Weak entities: weak entities are entities in a database schema that lack a primary key attribute of their own and instead rely on an identifying relationship with an owner entity for identification and existence.

 # Generalization in ER to RM involves:

1. Identifying common attributes.
2. Creating a table for the generalized entity.
3. Mapping specialized entities to separate tables.
4. Establishing primary key-foreign key relationships for inheritance.

5. Consolidating data from specialized entities into the generalized entity table.
6. Querying and retrieving data using joins between tables.

This process abstracts shared characteristics into a higher-level entity and maintains data integrity and structure in the Relational Model.

 # Normalization:

- **Normalization** is the process of improving a database schema to eliminate logical redundancy.
- It is based on the theory of functional dependencies (FD), which represent value-based constraints in the real world being modeled.
- Functional dependencies, such as A → B (where A determines B), help identify relationships between attributes.
- Examples include semester and course determining examiner, and student ID determining student name.

 # Anomalies in databases are situations that lead to inconsistencies or unintended consequences due to the structure or design of the database. Here's an explanation of each type of anomaly:

1. **Update Anomaly:**

    - Occurs when partial updates to the database result in inconsistencies due to data redundancy.
    - Example: Updating the name of a student with ID 23 to 'Liesl Gonzales' in the University table leaves the name for student with ID 42 unchanged, potentially causing inconsistency.

2. **Deletion Anomaly:**

    - Occurs when deleting data from the database leads to unintended loss of other related data.
    - Example: Deleting the record for student with ID 23 from the University table results in the loss of information about the course 'ADM', even though other students might be enrolled in it.

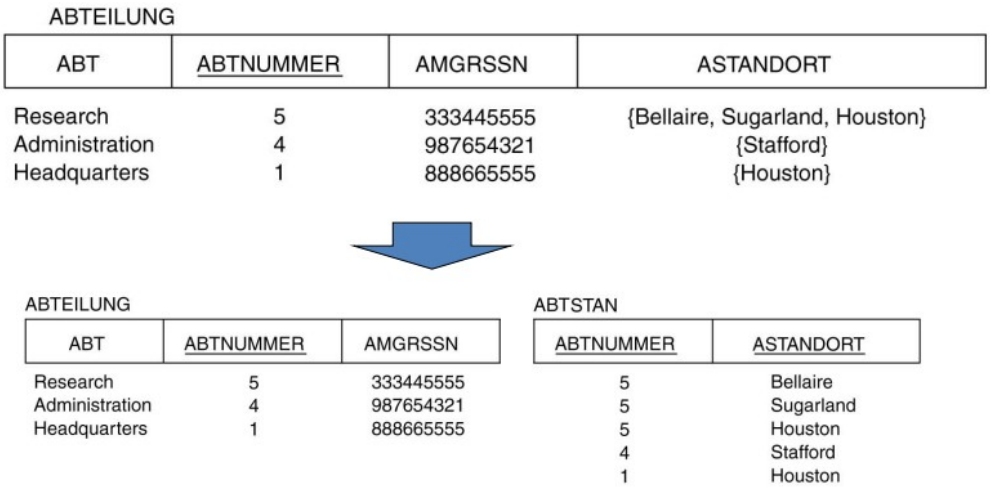
3. **Insertion Anomaly:**

    - Occurs when it is not possible to add data to the database due to the absence of related data.
    - Example: In a situation where there are no students yet, it's not possible to add a new lecture or room to the database because the presence of students is a prerequisite for such additions.

 # Normal Form:

1. First Normal Form (1NF):

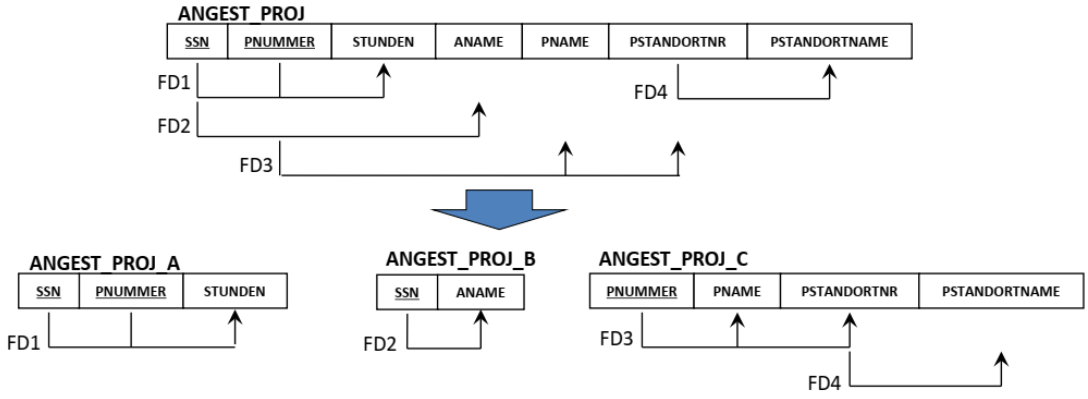
- Attributes contain only simple values, no lists or nested structures.
- Multi-valued attributes are moved to separate tables.



2. Second Normal Form (2NF):

- Database is in 1NF.
- In tables with composite keys, no non-key attribute depends on only part of the key.

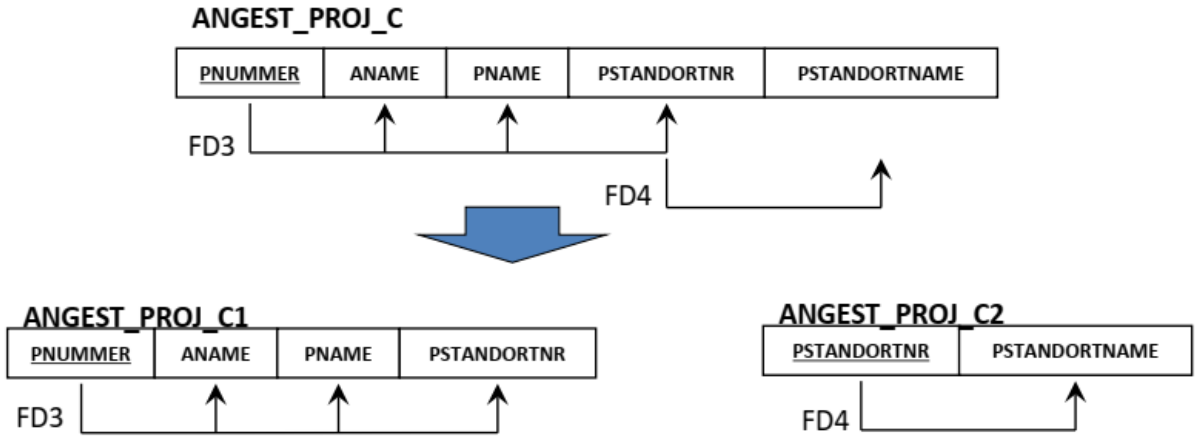
- Attributes dependent on part of the key are moved to new tables with the key's determining part.



3. Third Normal Form (3NF):

- Database is in 2NF.
- No non-key attribute is transitively determined by the key.

- Attributes transitively dependent on the key are moved to new tables with the determining attribute as the new key.



These normal forms help to eliminate redundancy and dependency issues in the database schema, ensuring data integrity and facilitating efficient querying and maintenance.

 # Relational algebra: *Relational algebra is a system of operations performed on tables, ensuring correctness and efficiency in relational database management systems.* Tables are objects, and operations produce tables as output, guaranteeing both accurate query results and efficient data processing.

The operations of relational algebra include:

1. **Selection (σ)**: Selects rows from a table that satisfy a specified condition.
2. **Projection (π)**: Selects specific columns from a table while retaining unique rows.
3. **Union (∪)**: Combines two tables to produce a table containing all unique rows from both tables.
4. **Intersection (∩)**: Produces a table containing rows common to both tables.
5. **Difference (−)**: Returns rows present in one table but not in another.
6. **Cartesian Product (×)**: Combines each row of one table with every row of another table, resulting in a new table with all possible combinations.
7. **Join ( )**: Combines related rows from two tables based on a common attribute.
8. **Division (÷)**: Returns all rows from one table that match every row in another table.

Lossless Join: ***An Equijoin is called lossless if all rows of R and S participate in the join. (Table name: R and S)***

$$Result = Reader \bowtie_{ReaderID=ReaderID} BookLending$$

Result	ReaderID	Name	BookID	ReaderID	ReturnDate
	205	Peter	1002	205	25-10-2016
	205	Peter	1006	205	27-10-2016
	207	Laura	1004	207	31-10-2016

Recreate by projection:

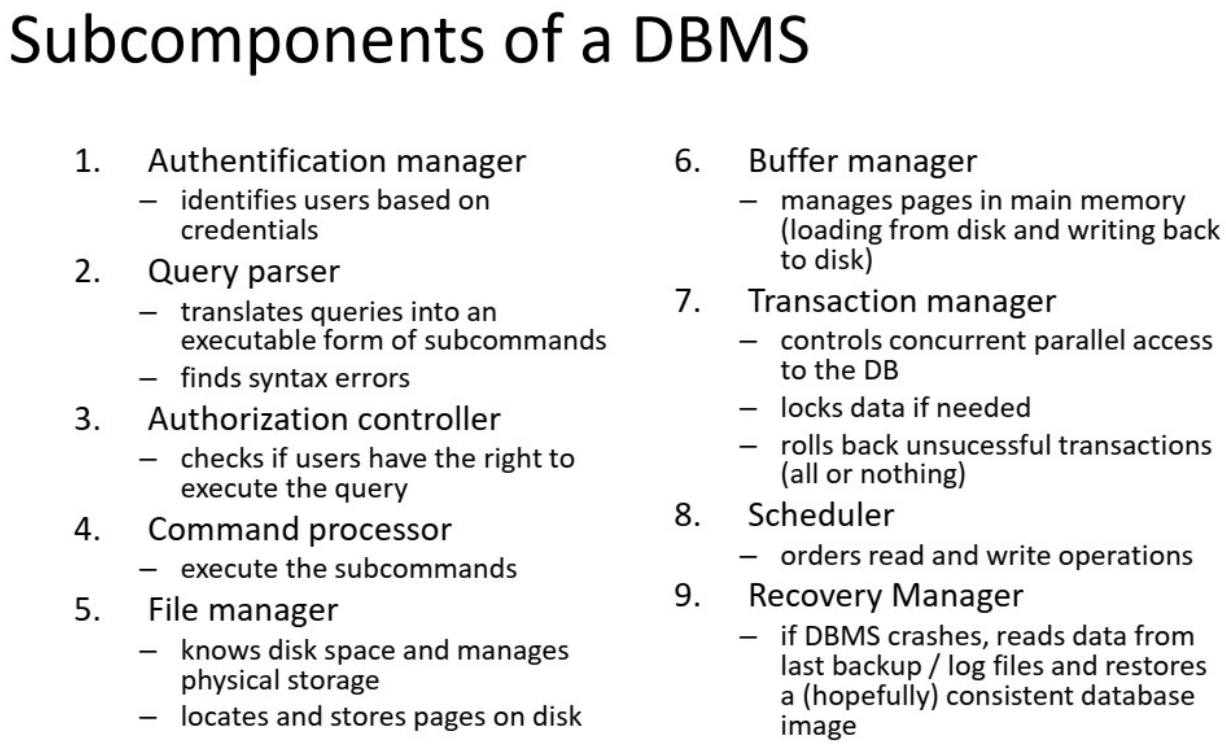
$$Reader = \pi_{ReaderID,Name} Result$$

$$BookLending = \pi_{BookID,ReaderID,ReturnDate} Result$$

Reader	ReaderID	Name
	205	Peter
	207	Laura

BookLending	BookID	ReaderID	ReturnDate
	1002	205	25-10-2016
	1006	205	27-10-2016
	1004	207	31-10-2016

Query execution: The query parser reads the SQL query and translate it into an executable from ***sub-commends***. For these it uses the relational algebra to create an algebra tree.



DBMS Sub-components

 # Strengths and weakness of RDBMS

Strengths of RDBMS:

- Mature technology with ACID properties.
- Good support for structured data.
- Wide developer adoption and vendor diversity.

Weaknesses of RDBMS:

- Designed for infrequent updates, leading to overhead.
- Variability in SQL dialects.
- Limited support for non-standard data types.
- Declarative access only, with short-lived transactions.
- Lower throughput compared to NoSQL.
- Rigid schema and lack of versioned data.

Extensible Records Stores

 # Logical Data model: It uses data duplication to improve performance. It begins with identifying entities and their relationships, then tailors table designs to support anticipated queries efficiently,

considering factors like query load and access patterns. This approach aims to strike a balance between data organization and performance optimization.

 # Data representation

• Instead of tuples: key-value pairs
– value: value in table cell
– key: column / attribute name

• Repetition of column name for each value
– better use short column names!

• Flexible structure:
– for each row, the column names could be different

Example (ignores reader ID and book ID)

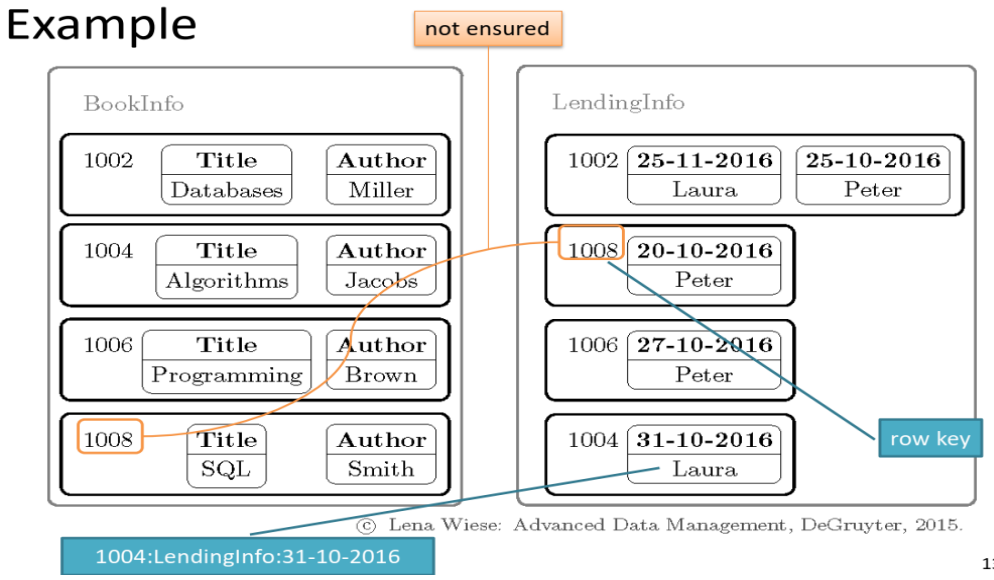
Title: Databases	Author: Miller	25-11-2016: Laura	25-10-2016: Peter
Title: Algorithms	Author: Jacobs	20-10-2016: Peter	
Title: Programming	Author: Brown	27-10-2016: Peter	
Title: SQL	Author: Smith	31-10-2016: Laura	

 # Column families:

• To further structure the key value pairs, column families group columns that are often used together.
– *Identified by column qualifier.*
• ***Column name = column familiy name + column qualifier***
• When processing a query, only affected column families are fetched into memory.
• New columns can be added to column families at runtime.

Row Key:

- Unique identifier for grouping columns of the same entity within a column family.
- Unique within each column family.
- Used for cross-referencing between column families, but no inherent referential integrity.
- To identify a specific cell combine, ***rowkey + column family + column qualifier.***



 # Time and Upserts in logical data model:

1. Time Consideration:

- Each insert or update is timestamped, enabling automatic versioning.
- Configurable options include time-to-live and maximum version retention.

2. "Upsert" Functionality:

- Combines insert and update operations.
- No distinction between insert and update; new versions are inserted.

These features streamline data management and querying in extensible record stores, enhancing version control and simplifying operations.

 # Physical storage:

→ Immutable Sorted Data Files:

- Data files are immutable and sorted.
- Follows a write-optimized storage model, allowing only appends.
- ***Once written, data files remain unchanged.***

→ Memtable:

- Each column family has a memtable.
- Collects the most recent writes with a fixed size and timestamp.
- For "upsert" operations, the memtable contains the value.
- For deletions, the memtable marks the value as empty using a **tombstone**.
- All older version will be invisible to the user.

Writing to memory tables and data files

- When memtable is full, data is written to a sorted data file
- Sorted files are immutable; when data in file is upserted later, newer files need to store that

 # Sorted Data Files:

- **Flush Operation**: Records from the memtable are sorted by key and written to sorted data files on disk. A new memtable is started for subsequent writes.
- **Immutable Nature**: Sorted data files are immutable once written.
- **Modification Handling**: Changes for a key are stored in later flushed files.
- **Advantages**: Simplifies buffer management with no "dirty pages" that contain modifications that have to be translated to writes on the on-disk records. And maintains chronological order with internal sequence numbers.

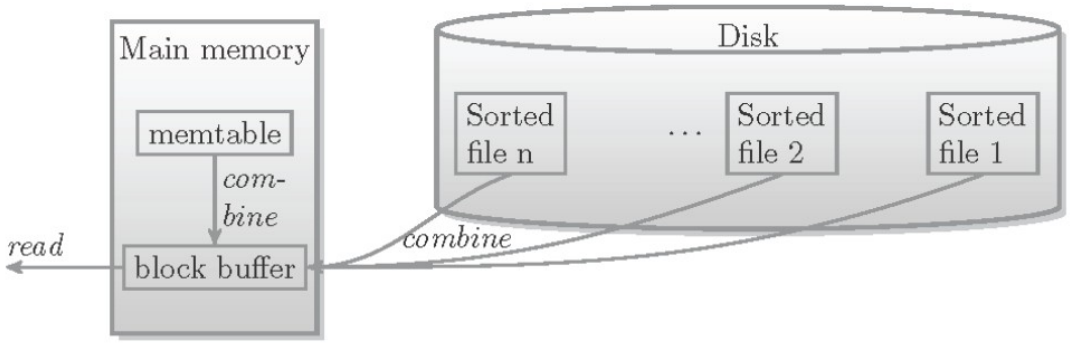
 # Tombstones:

• **Deletions** are treated by writing a new record for a key. However, this record has no value assigned to it; instead a delete marker (called tombstone) is attached to the record.

• **The tombstone** masks all previous versions (with timestamps prior to the timestamp of the tombstone) which will then be invisible to the user.

• **Types of tombstones** (differ in the scope of records that they mark as deleted):
– single version of a column (defined by its exact timestamp)
– entire column (with all its versions)
– entire column family (with all versions of all columns)

Reading from memory tables and data files



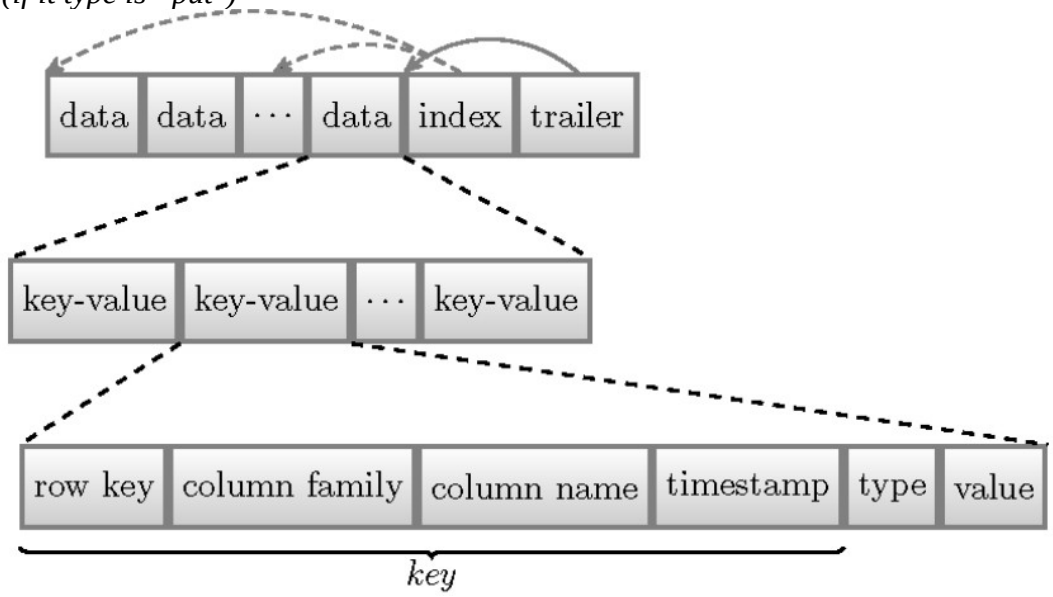
- Two types of read request:
  - get (AKA point query): get a particular row
  - scan (AKA range query): iterate over a contiguous range of rows (ordered scans are efficient)

Combining data for read is not trivial:

- Timestamp clashes arise with multiple records for the same key and version.
- Timestamps being part of keys complicate data retrieval.
- Appending new records to the memtable and flushing to data files can lead to reading challenges, resolved by unique sequence numbers for files.

 # File format: An on-disk data file is composed of several data blocks and a data block may contain one or more key-value pairs.

• Each key-value pair contains
– *row key*
– *column family*
– *column name*
– *timestamp*
– *type information:*
  • *put: value is an upsert*
  • *deletion: type of tombstone*
– *value (if it type is "put")*



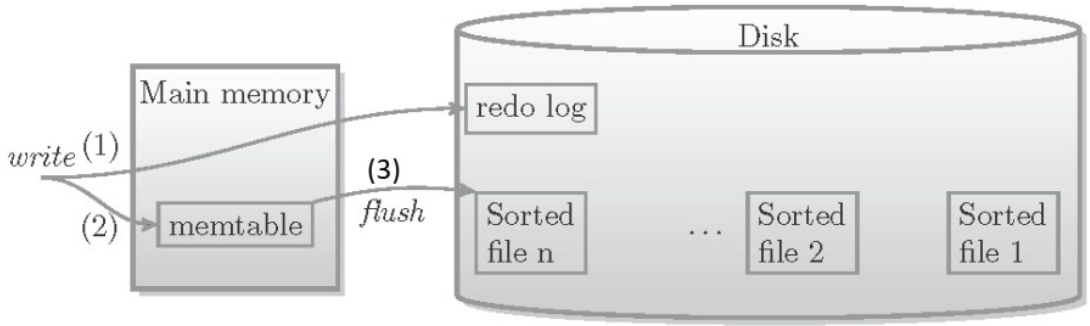
 # Redo Logging: Memtable keep all the data in volatile memory until it is eventually flushed to disk. In case of data loss Redo logging comes in to the picture.

→ Logging process:

– An on-disk log file keeps track of all records that are appended to the memtable but have not yet been flushed to the disk

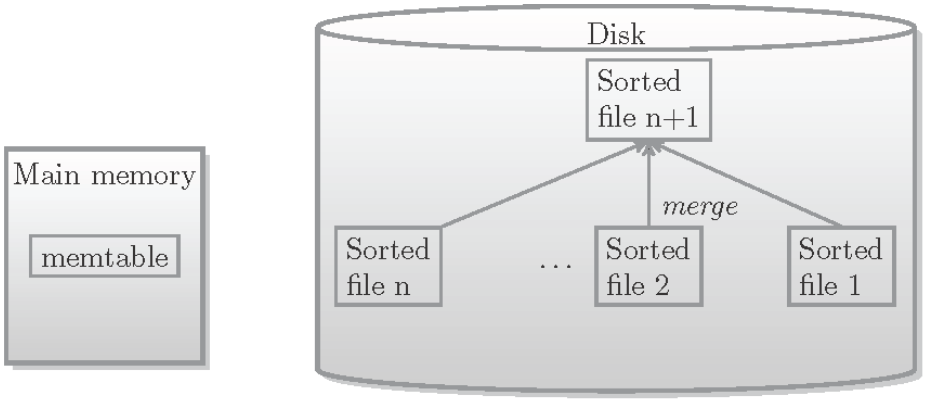
– This means that all data have to be written twice: once to the log file and then to the memtable

– Each record receives a log sequence number (LSN) for redo logging

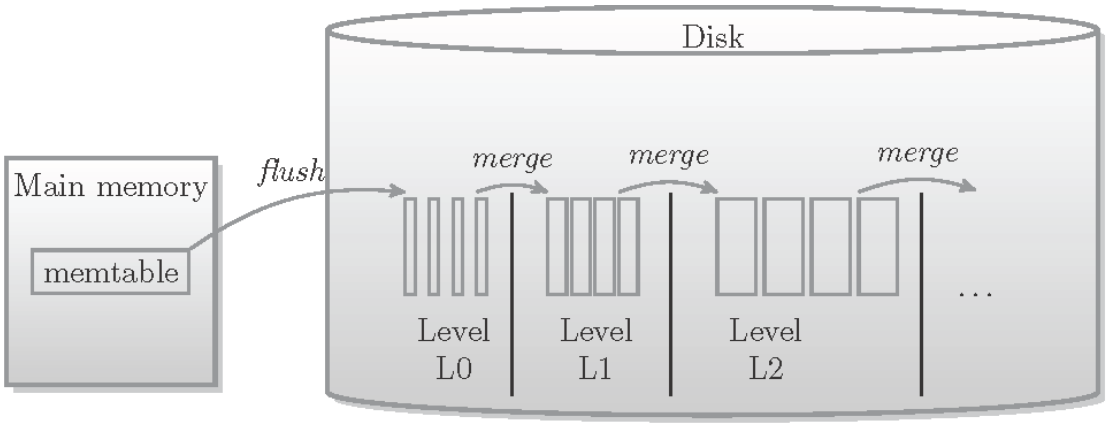


 # Compaction: Its a process to remove any unwanted records and merge a set of data files into a new one. In the compaction process:

- **Sorting**: All key-value pairs are sorted.
- **Index Rebuilding**: The index is reconstructed.
- **Data Cleanup to ignore all outdated data**:
    - **Time-to-Live (TTL)**: Outdated versions are ignored based on TTL values.
    - **Tombstones**: Older versions and tombstones are disregarded.
    - **Maximum Versions**: Older versions exceeding the maximum specified are removed.
- **Types**:
    - **Minor Compaction**: Merges a small subset of data files.
    - **Major Compaction**: Combines all data files into a new one.



 # Leveled compaction is a method for managing data compaction in databases efficiently. It organizes data files into levels, with lower levels containing smaller files. New data is written to the lowest level, and compaction moves data up one level at a time to prevent duplication. This approach optimizes compaction performance and reduces unnecessary data processing.



© Lena Wiese: Advanced Data Management, DeGruyter, 2015.

 # Bloom Filter: Bloom filters are employed in extensible record stores to efficiently determine set membership, especially when searching for specific records. They offer a rapid way to confirm if a record is not included in a set, aiding quick decision-making during searches. When searching for a key, the Bloom filter is accessed first; if the key is not found, other data files are checked.

There four types of errors in a Bloom filter:

1. **True Positive**: The Bloom filter correctly identifies an element as a member of the set, and it is indeed present in the set.
2. **False Positive**: The Bloom filter incorrectly identifies an element as a member of the set, even though it is not actually present in the set.
3. **True Negative**: The Bloom filter correctly identifies an element as not being a member of the set, and it is indeed absent from the set.
4. **False Negative**: The Bloom filter incorrectly identifies an element as not being a member of the set, even though it is actually present in the set.

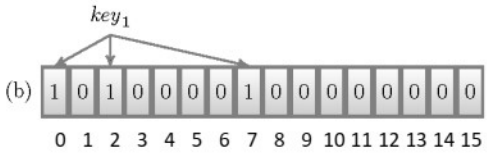
The key characteristic of a Bloom filter is that it can only produce false positives; **false negatives are not possible if the Bloom filter** is properly constructed and utilized.

→ **Collision:** It occurs when two different input value c and c' are mapped to the same value (hi (c) = $hi(c')$). **This situation can lead to false positives.**

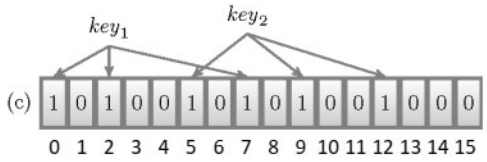
Example

a) Initialization: all values are 0

b) Insert key_1:
$h_1(key_1) = 0, h_2(key_1) = 2, h_3(key_1) = 7$

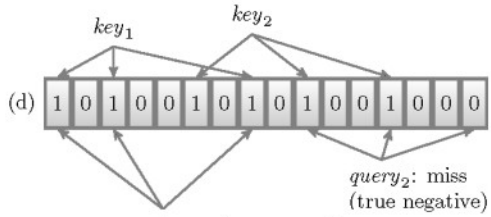


c) Insert key_2:
$h_1(key_2) = 5, h_2(key_2) = 9, h_3(key_2) = 12$



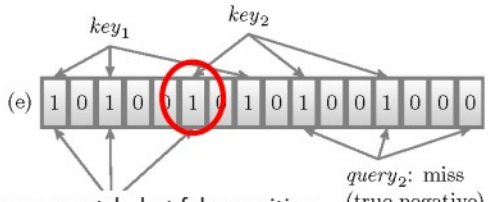
d) $query_1$: search for key_x where
$h_1(key_x) = 0, h_2(key_x) = 2, h_3(key_x) = 7$

$query_2$: search for key_y where
$h_1(key_y) = 9, h_2(key_y) = 12, h_3(key_y) = 15$



$query_1$: match, and true positive
because $key_x = key_1$)

$query_3$: $h_1(key_z) = 0, h_2(key_z) = 2, h_3(key_z) = 5$



$query_3$: match, but false positive,
because $key_z \neq key_1, key_2$

Implementation:

 # Apache Cassandra is a distributed NoSQL database management system designed to handle large amounts of data across multiple commodity servers, providing high availability and fault tolerance.

→ **Partitioning:** Data in Cassandra is partitioned across the cluster using a consistent hashing algorithm. Each row is assigned to a specific node in the cluster based on the hash of its partition key. This allows Cassandra to evenly distribute data across nodes in the cluster and provides horizontal scalability.

→ **Replication:** Cassandra provides built-in replication to ensure high availability and fault tolerance. Data is replicated across multiple nodes in the cluster, with each replica stored on a different node. Cassandra supports configurable replication strategies, allowing users to define how many replicas of each piece of data should be stored and on which nodes they should be placed.

Cassandra by example

- Create a keyspace with certain settings for replication:

```
CREATE KEYSPACE library
  WITH REPLICATION = {
  'class' : 'SimpleStrategy',
  'replication_factor' : 3};
```

- Create a "table" (Cassandra term for column family):

```
CREATE TABLE bookinfo (
  bookid int PRIMARY KEY,
  title text,
  author text);
```

- Insert some data:

```
INSERT INTO bookinfo (bookid, title, author)
  VALUES (1002, 'Databases', 'Miller');
```

- Create and index to enable filtering on other column values:

```
CREATE INDEX ON bookinfo (title);
CREATE INDEX ON bookinfo (author);
```

Distributed Data Management

 # Features of a distributed database management system (DBMS):

1. **Load Balancing**: Distributes processing load evenly among servers to prevent hotspots and ensure efficient resource utilization.
2. **Flexible Scalability (Elasticity)**: Allows servers to dynamically join and leave the network, accommodating changes in workload or system capacity. Also known as "membership churn."
3. **Heterogeneous Nodes**: Supports nodes with varying capabilities, allowing for a mix of hardware configurations within the system.
4. **Symmetric Configuration**: All nodes are configured identically, enabling any node to replace a failed node seamlessly. This ensures high availability and fault tolerance.
5. **Decentralized Control**: Utilizes peer-to-peer algorithms for data management, eliminating single points of failure and improving fault tolerance. This decentralized approach enhances system reliability and resilience.

 # Distribution Transparency in a Distributed Database Management System (DDBMS) ensures that users perceive the system as a single, centralized database, hiding the complexities of data distribution. It includes:

1. **Access Transparency**: Users interact with the system through a uniform query and management interface, regardless of the underlying distribution.
2. **Hidden Aspects**: Users must be unaware of:
    - Data distribution and location.

- Replication details.
- Data fragmentation specifics.
- Data migration.
- Concurrent user access.
- System failures and their impacts.

 # Failures in Distributed Systems:

1. **Server Failures**:
    - Processing failure: Server fails to process a message.
    - Crash: Server crashes and requires restart.
    - Processing delay: Servers experience delays in message processing.
    - Incorrect messages: Servers send incorrect messages.
2. **Message Failures**:
    - Loss or delay: Messages may be lost or delayed during transmission.
3. **Link Failures**:
    - Broken or corrupt link: Communication links between servers are broken or corrupt, leading to message failures.
    - Duplicate messages: Links may duplicate messages, causing confusion.
4. **Network Partition**:
    - Split in the network: Link or server failures lead to a split in the network.
    - Partial communication: Servers in each partition can communicate internally but are unable to reach servers in the other partition.

 # Fragmentation: It is a division of data across different storage. A good fragmentation is the one that supports workload.

→ ***Properties of workload to be considered:***
– Type of accesses (mostly reads or mostly writes?)
– Access patterns (hot data, cold data)
– Affinity of records (which data is accessed together)
– Frequency of access (how often?)
– Duration of access (long term, short term?)

 # A good fragmentation strategy in a distributed database ensures:

1. **Completeness**: No data loss occurs during fragmentation.
2. **Soundness**: No additional data are introduced when original data is reconstructed.
3. **Data Locality**: Data accessed together is placed in the same fragment, enhancing performance.

4. **Minimal Communication Cost**: Data movement between servers is minimized, reducing network overhead.
5. **Efficient Data Management**: Fragments are small, enabling fast query processing.
6. **Load Balancing**: Workload is evenly distributed across servers, avoiding hotspots and bottlenecks.

 # Types of Fragmentation:

1. **Handmade/Manual/User-defined**: Database admins define fragmentation.
2. **Random**: Data is partitioned randomly.
3. **Structure-based**: Fragmentation is based on the data schema definition.
4. **Value-based**: Fragments are determined by data values, often using selection predicates or minterms.
5. **Range-based**: Fragments are specified by ranges of values.
6. **Hash-based**: Fragmentation is determined by hashing data record keys.
7. **Cost-based**: Fragmentation minimizes overall cost, considering factors like communication and efficiency.
8. **Affinity-based**: Fragmentation considers how often data is accessed together, such as through JOINs.
9. **Clustering**: Fragments are based on coherent data structures, often identified by clustering algorithms.

Data Allocation

→ Data Allocation Strategies:

1. **Range-based Allocation**:

   - Based on range-based fragmentation.
   - Divides data based on specified value ranges.
   - May lead to load imbalance if a server receives a disproportionately popular range of values (e.g., all entries for a trending topic).

2. **Hash-based Allocation**:

   - Utilizes a hash function over input fragments.
   - Often paired with hash-based fragmentation.
   - Consistent hashing is preferred as it avoids re-hashing the entire dataset when data grows.

3. **Cost-based Allocation**:

   - Frames data allocation as an optimization problem, aiming to minimize costs associated with communication, storage, and processing.

 # Hash-based Allocation: Hash-based allocation uses a hash function to distribute data among servers. Each data item is hashed, resulting in a unique value, which determines the server it is stored on.

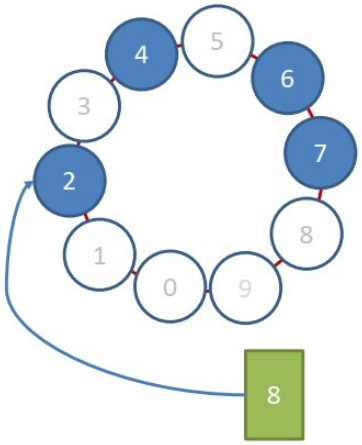
For example, student records with unique IDs are hashed to specific servers, ensuring even distribution and efficient retrieval.

 # Consistent hashing is a technique used in distributed systems to distribute data across a set of servers in a scalable and efficient manner. The main idea behind consistent hashing is to minimize the amount of data movement required when servers are added or removed from the system.
In consistent hashing:

1. **Hash Ring:** Servers are arranged in a circular hash ring, where each server is assigned a unique position on the ring.
2. **Data Mapping:** Data items are also hashed, resulting in a hash value that corresponds to a position on the hash ring.
3. **Server Assignment:** Each data item is assigned to the server whose position on the hash ring is the closest in a clockwise direction to the hash value of the data item.
4. **Load Balancing:** Since the hash ring is circular, adding or removing a server affects only the immediate neighbors on the ring. This minimizes the amount of data that needs to be moved when servers are added or removed, making consistent hashing highly scalable and efficient.

→ How data Stores:

– Hash the key of the data (e.g.: 8)
– Walk the ring clockwise to find a server which position is larger
– Wrap around if necessary
– Store the data on the next server (here: 2)



If new server gets active it can take over parts of the ring
– Only neighboring nodes are affected
– If a server with H(IP) = 9 joins the ring, it takes
over data from 8 to 9 from node H(IP) = 2

 # Consistent Hashing with Virtual Nodes:

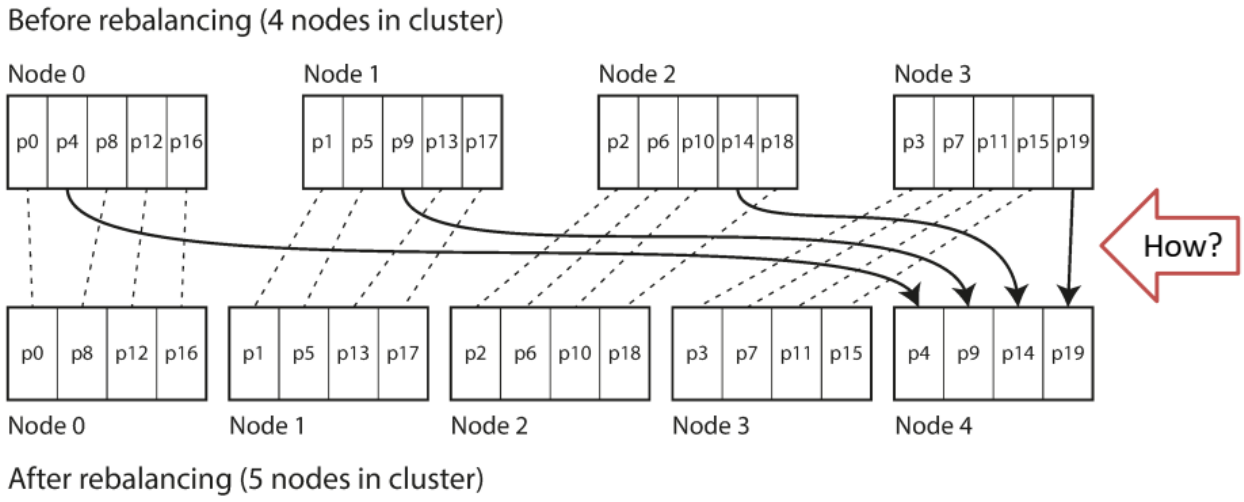
Issue: Basic consistent hashing leads to uneven data and load distribution, and doesn't account for node performance differences.

Solution: Introduce Virtual Nodes (VNs):

VNs mimic real nodes and represent positions on the hash ring.
Each real node can manage multiple VNs, providing flexibility.

Virtual Nodes improves load distribution and scalability in distributed systems by introducing virtual nodes. Real nodes manage multiple virtual nodes, ensuring even data distribution. When adding or removing nodes, virtual nodes facilitate efficient load balancing without disrupting the system.

 # Rebalancing in distributed systems, using the "Fixed number of partitions" strategy, allows new nodes to acquire partitions from existing nodes. The total number of partitions remains constant, only the assignment of partitions to nodes changes. Determining which partitions to transfer depends on specific metrics and balancing goals.



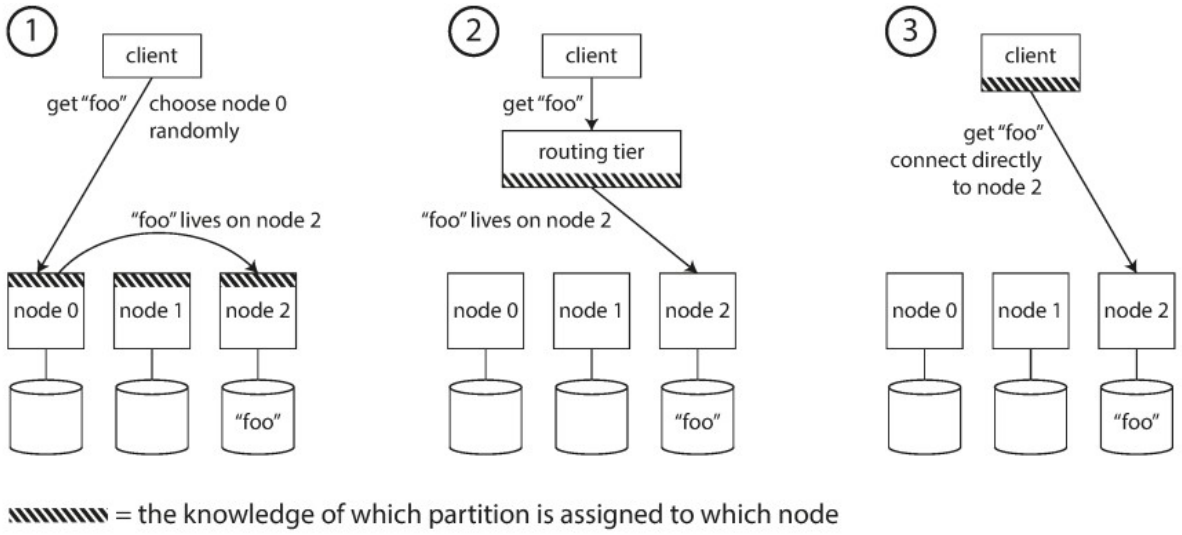
There are three types of re-balancing:

Fully automated:
– System determines when it‘s needed and how
– Can be unpredictable: Expensive operation, could lead to overload of system when you need it least

Fully manual:
– DB admin decides and explicitely configures it
– Needs a lot of insight and information to do it right

Semi-automated:
– System suggests re-balancing
– DB admin reviews and commits

 # Request Routimg: If hashing does not lead directly to the node, there are three ways to inform client about data location,



Discussion:

1. Distributed knowledge about partition location
– Needs protocol to distribute the knowledge (e.g., Gossip protocol in Cassandra)

2. Coordination service
– Needs extra service to manage that service (e.g., Zookeeper)
– Can become a single point of failure

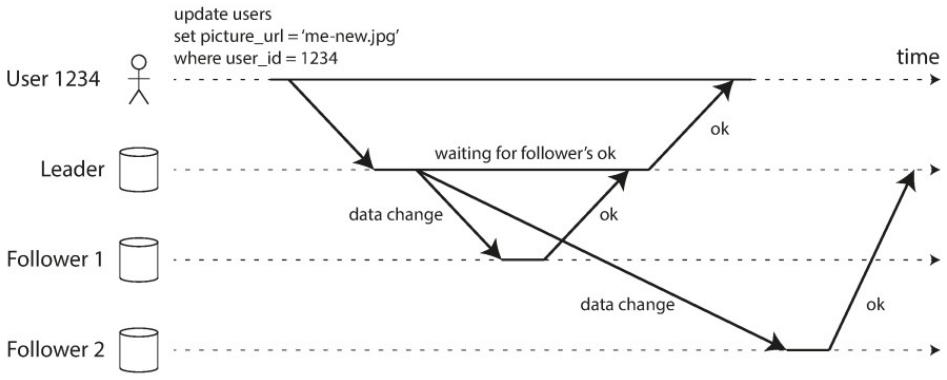
3. Local knowledge at client
– Violates Location Transparency and Migration Transparency
– Can be used as cache: If it fails, use 1 or 2 to get the right location

 # In distributed systems, replicas are organized into Leader nodes and Follower nodes:

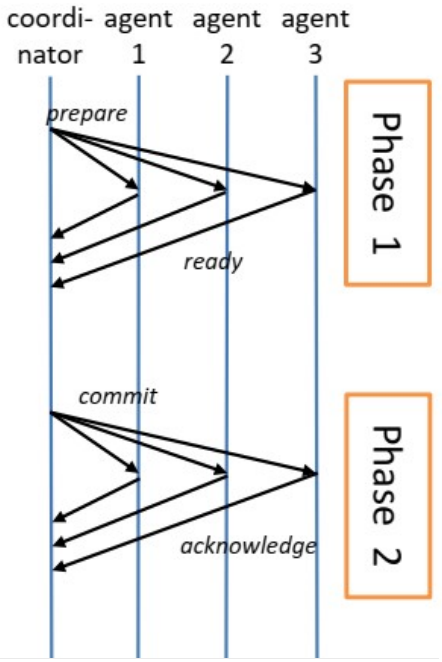
- **Leader (or master)**: Handles write operations and forwards them to all followers.
- **Follower (or read replicas)**: Supports read operations and remains synchronized with the leader.
- **Read Operations**: Can be served by any replica.
- **Write Operations**: Directed to the leader, which propagates changes to all followers to maintain consistency.

  Having a single leader poses a single point of failure, hence multiple leaders are synchronized to ensure fault tolerance and reliability.

 # Synchronous vs asynchronous followers:



 # Two-Phase Commit:



Two-Phase Commit (2PC) is a widely used protocol for coordinating distributed operations:

1. **Phase 1 (Voting)**: Coordinator asks all participants to "prepare" for the operation. Participants respond with "ready" if they can proceed.
2. **Phase 2 (Decision)**: Coordinator sends a "commit" message to all participants if all are ready. Participants respond with "acknowledge_commit" to confirm they can commit.
3. **Optional Abort Phase**: If any participant cannot commit, the coordinator sends an "abort" message to all participants who voted "ready". They must rollback their changes. The coordinator may restart the protocol.

2PC ensures that all participants agree on the outcome of the operation, but it can be blocked by failures or network partitions.

 # Limitations of 2PC:

Issues with Two-Phase Commit (2PC) include:

1. **Scalability**: Not efficient with a large number of agents. A single late or non-responding agent can cause the entire operation to abort.
2. **In-Doubt State**: The period between Phase 1 and Phase 2, where the coordinator has received "ready" responses but hasn't yet committed, is called the "in-doubt state". If the coordinator fails irrecoverably during this time, clients are left unable to proceed with either committing or aborting the operation, leading to potential data inconsistency.

 # Paxos Algorithms:

Paxos algorithms are a group of **protocols** designed to achieve consensus among unreliable processors in a network, such as a cluster of databases. Basic Paxos can handle various non-Byzantine failures like crashes, message loss, and reordering.

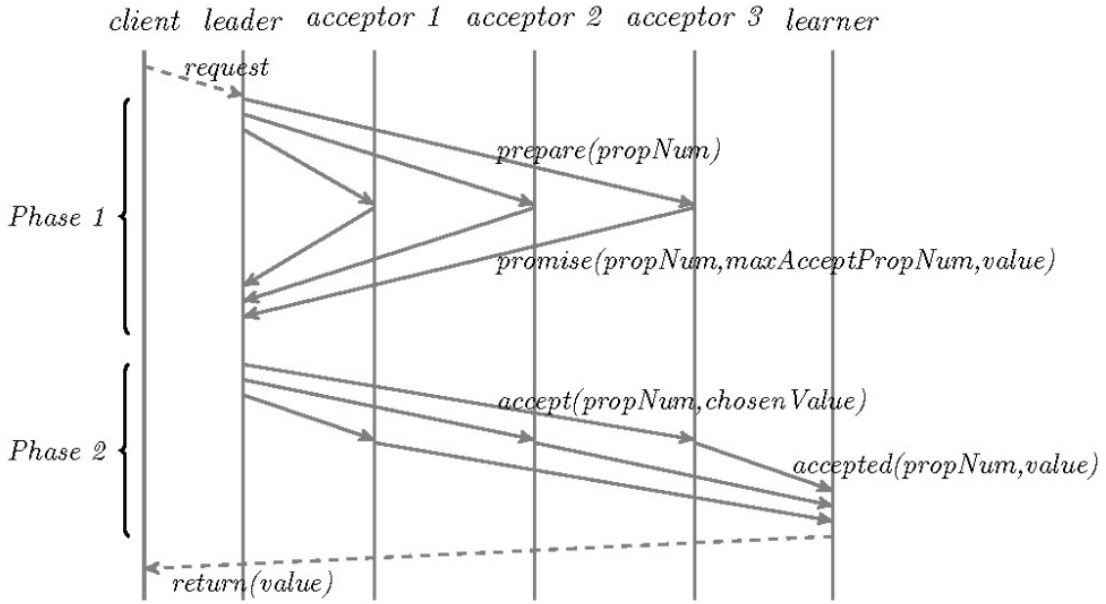
Basic Paxos involves four roles for agents:

1. Proposer: Initiates consensus, assigns unique proposal numbers, and requests responses from acceptors.
2. Leader: Elected from proposers, oversees the process for a client request.
3. Acceptor: Accepts proposals based on value and proposal number, storing the last accepted proposal number permanently.

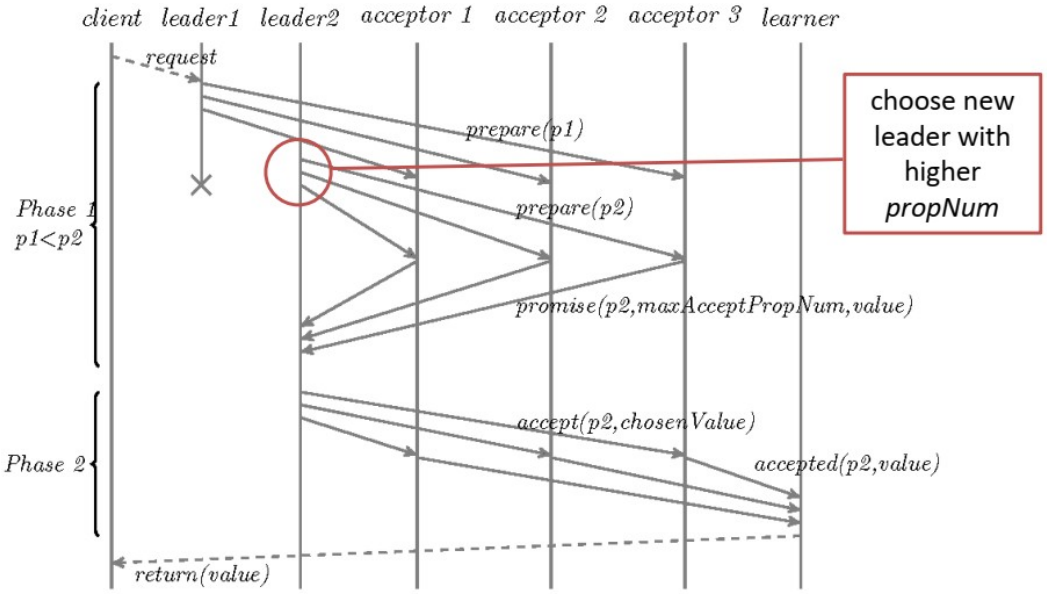
4. Learner: Receives accepted values from acceptors, chooses the consensus outcome, and returns it to the client.

Basic Paxos involves two phases:

1. Read phase (Phase 1):
    - Leader prepares for consensus by sending a prepare message with a proposal number to acceptors.
    - Acceptors respond with promise messages containing their current state.
    - Leader selects a value based on the highest proposal number received.
2. Write phase (Phase 2):
    - Leader sends the chosen value to acceptors with an accept message.
    - Acceptors notify learners of the chosen value with accepted messages.
    - For consensus, the learner must receive the value from a majority of acceptors.



Without Failure



During leader failure

 # Fundamental properties of the Basic Paxos algorithm:

Nontriviality: Only proposed values are learned.
Stability: Learners only learn one value or none.
Consistency: All learners converge on the same value.
Liveness: Eventually, a proposed value will be learned, ensuring progress.

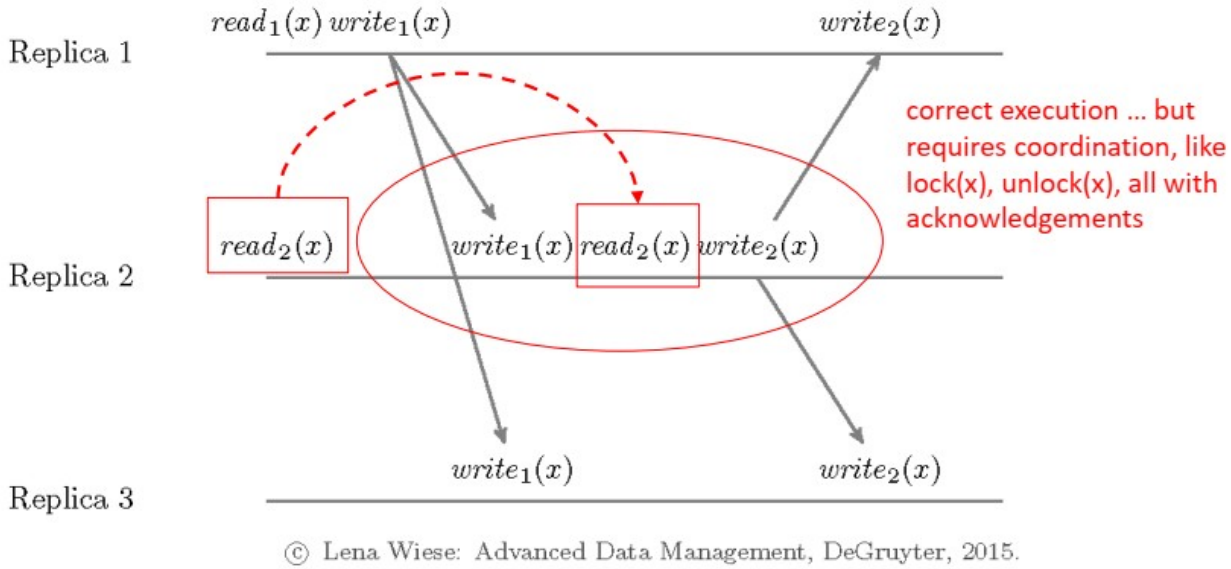
 # Paxos variants:

In Cheap Paxos, auxiliary acceptors are added to reduce message overhead, operating only when main acceptors fail temporarily.

Generalized Paxos allows partial order on commutative operations, ensuring consensus on sequences of operations rather than individual ones. e.g., when three client wants to add their numbers to a value: acceptors can execute this in any order, result will be the same.

 # Consistency:

→ **As in ACID properties**: Before and after a transaction, all the DB constraints are full-filled like internal integrity.
→ **In distributed databases (and in CAP theorem)**, consistency extends to ensuring all clients have the same data view across replicas, ideally with immediate and ordered updates, albeit with potential coordination overhead, leading to varying consistency levels in distributed systems.



 # Quorums: Flexible mechanism to avoid stale reads and lost updates in distributed DBMS. A (read/write) quorum is a subset of replicas that has to be contacted for a operation (read/write) and needs to acknowledge before that operation is sucessful.

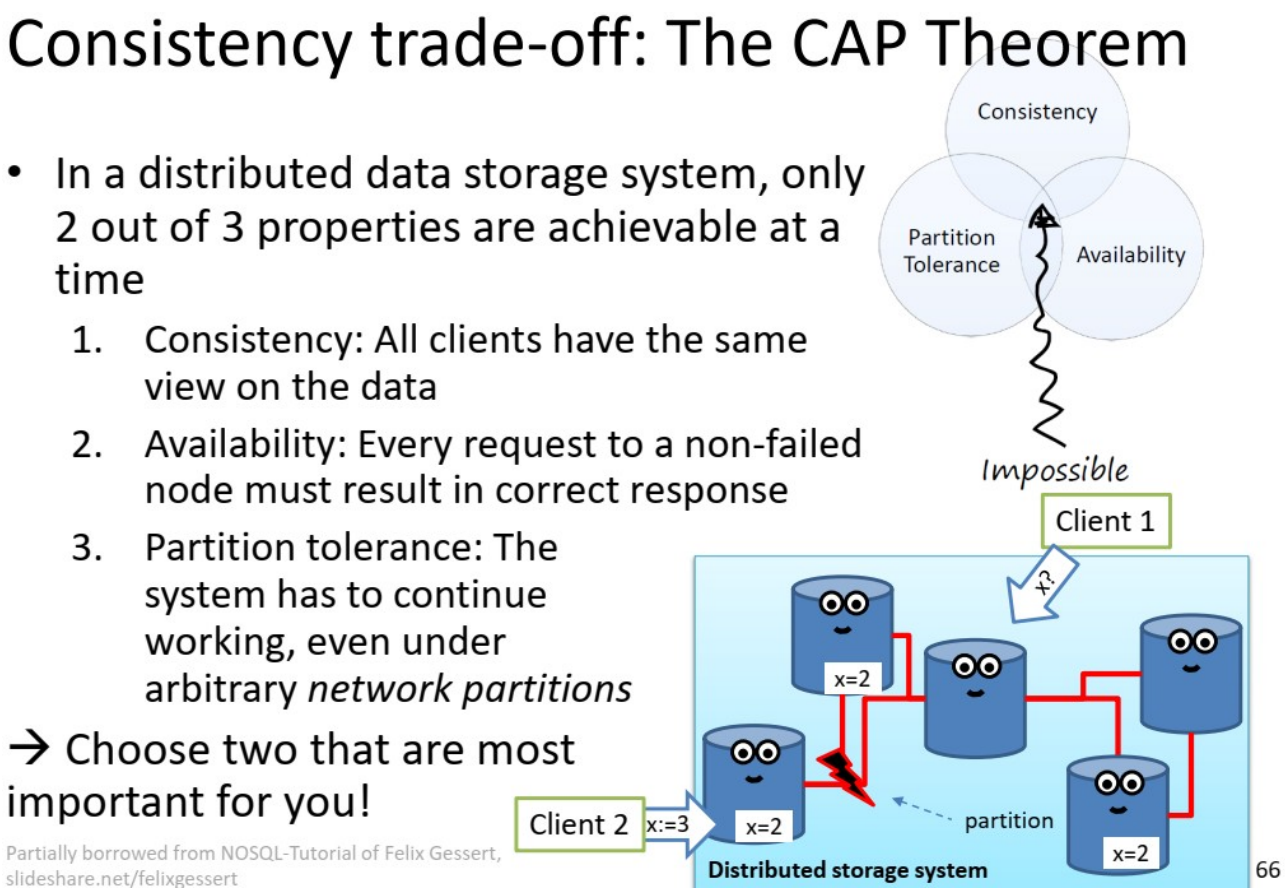
$$R + W > N$$

→ **Benefit:** Quorums ensure consistent write order and data availability. They reduce latency and tolerate network partitions if the required quorum is reachable.

 # Eventual consistency refers to the concept where, in a distributed system, all replicas will eventually converge to the same state after a certain period of time, allowing for temporary inconsistencies between replicas.

On the other hand, client-centered consistency models prioritize the user's perspective, ensuring consistency within a single user session, often offering different levels of guarantees tailored to the user's interaction with the system.

 # CAP Theorem:



 # NoSQL Decision Tree:

1. Access:

- Fast Lookups: Consider key-value stores or document databases.
- Complex Queries: Look into relational databases or column-family stores.

2. Volume:

- RAM: Opt for in-memory storage.
- HDD-Size: Choose databases supporting file-based storage or RAID.

- Unbounded: Go for distributed systems for handling big data.

3. CAP Theorem:

- Consistency: Decide between strict consistency, availability, or partition tolerance.
- Availability: Prioritize response correctness over consistency or partition tolerance.
- Partition Tolerance: Ensure system operation even during network partitions.

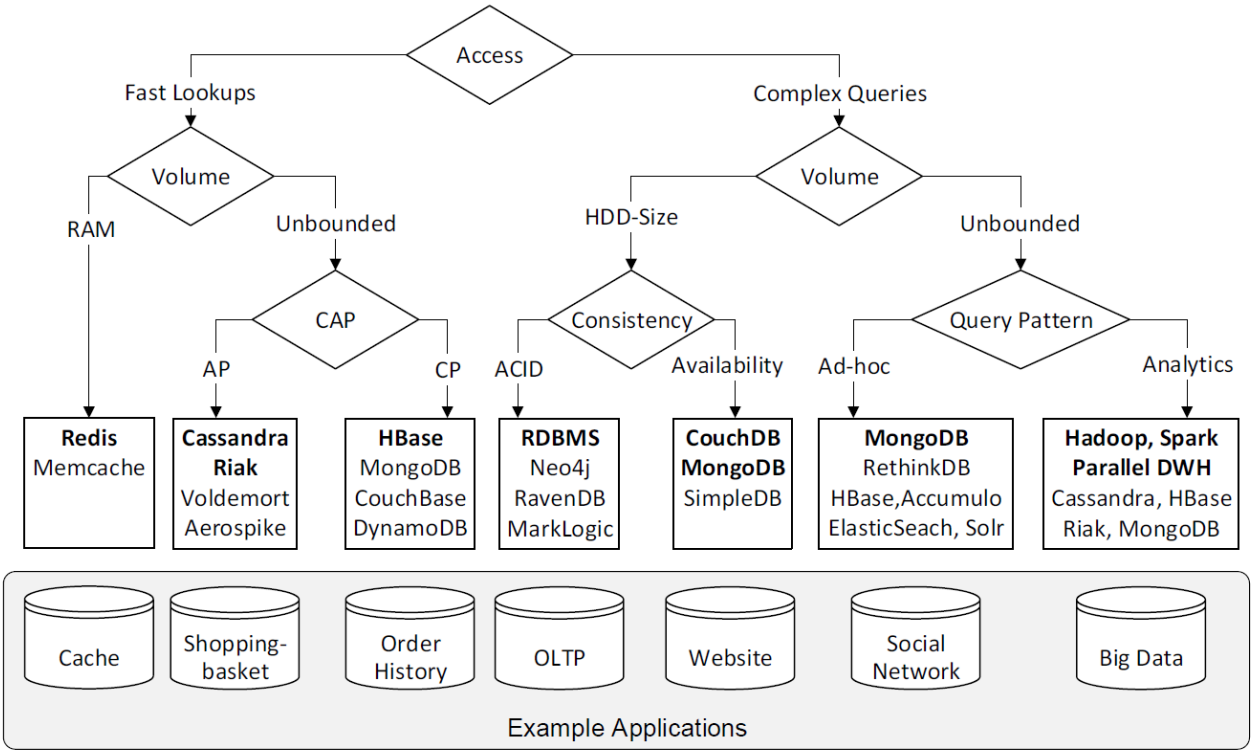
4. Consistency:

- ACID: Assess the need for full transactional properties.
- Availability: Balance consistency with response time.

5. Query Pattern:

- Ad-hoc: Consider databases with flexible schema and dynamic querying.
- Analytics: Opt for databases with robust analytics capabilities and efficient query processing.

Choosing the right NoSQL database involves balancing trade-offs based on these factors to meet your specific use case requirements.



Document Database

 # Popular document formats include:

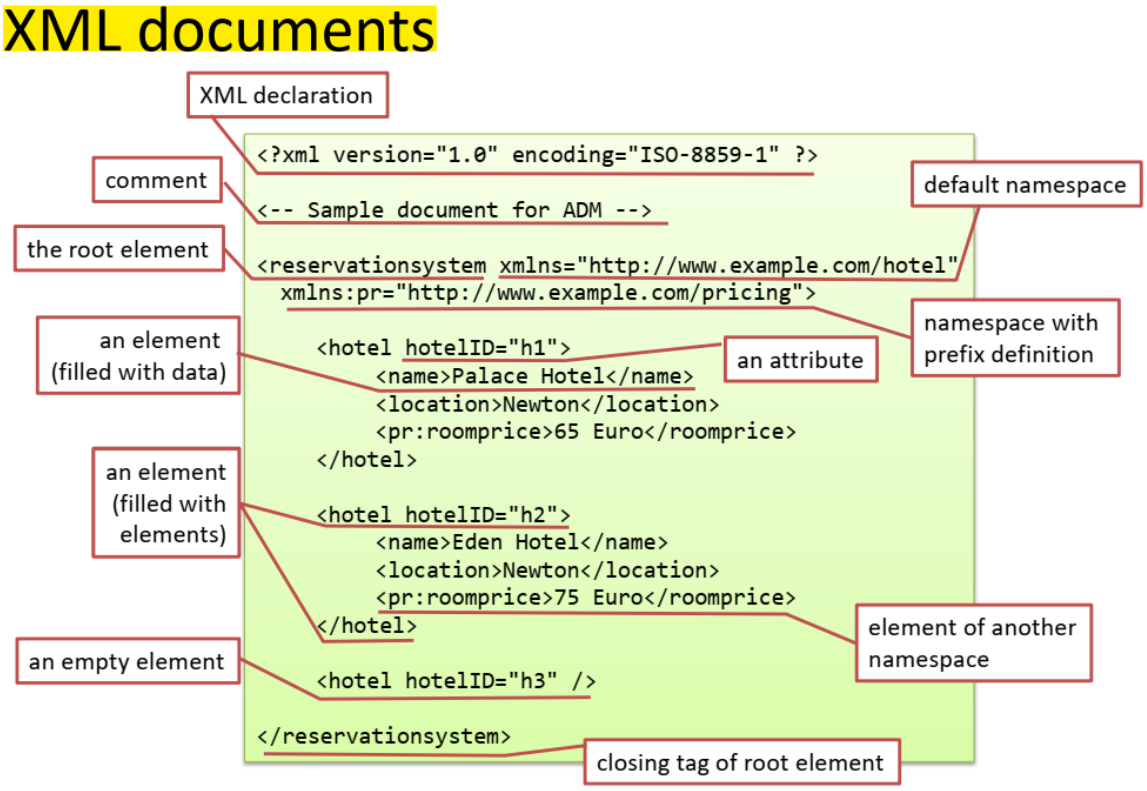
1. XML (Extensible Markup Language):

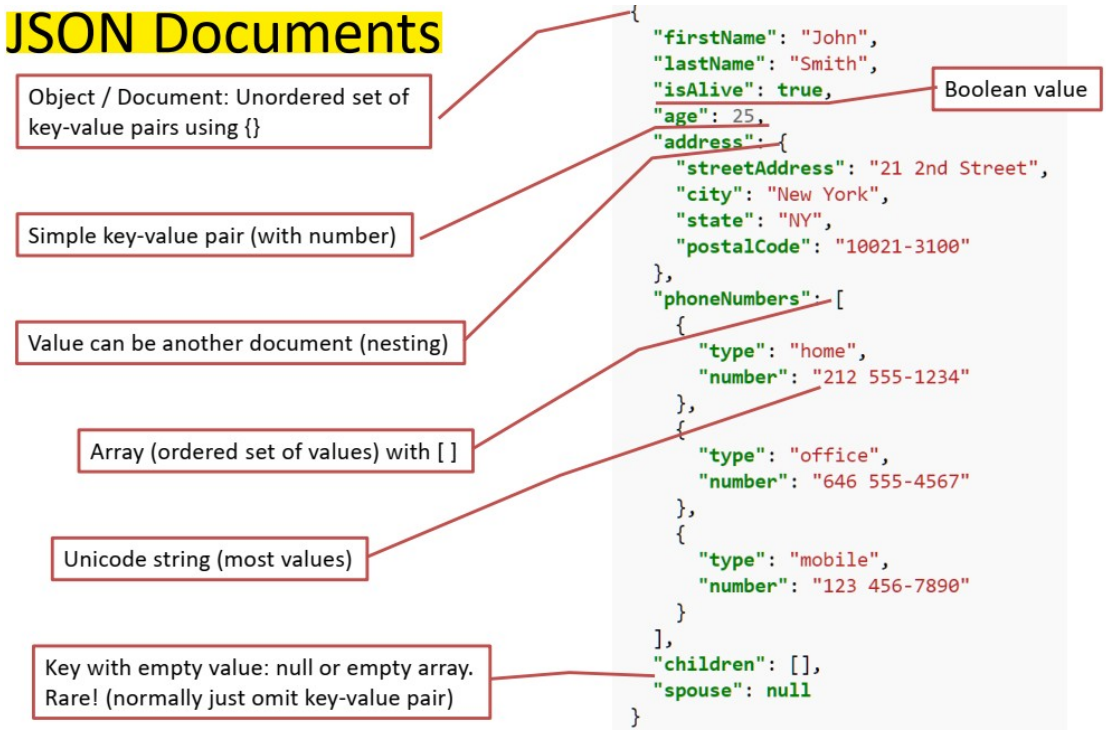
- Widely used with a large set of open standards.

- Supports schema languages like XML Schema and DTD.
- Queryable using XPath, XQuery, or integrated into SQL.
- Numerous domain-specific document standards available.

2. JSON (JavaScript Object Notation):

- Standardized in 2013 by Ecma International.
- Based on a subset of JavaScript.
- Supports schema definition with JSON Schema.
- Queryable using JavaScript.





 # When to use embedded data models:

- For "contains" relationships and one-to-many relationships.
- When read performance is crucial.
- Embedded models avoid data duplication.

When to use normalized data models:

- To prevent data duplication that would not enhance read performance.
- For complex many-to-many relationships.
- For modeling large hierarchical data sets.

 # Mongo DB: Schema-free document database with tunable consistency, which….

• Allows complex queries and indexing
• Supports sharding and replication
• Storage Management:
– Write-ahead logging for redos (journaling)
– Storage Engines: memory-mapped files, in-memory, …
• Represents documents with JSON, stores them in BSON

→ **BSON:** It is binary serialization of JSON object. Most interesting feature is fast-traversebility.

 # MongoDB's storage model:

1. **Ranged Sharding**:
    - Documents are divided across shards based on the value of a shard key.
    - Optimizes queries that involve ranges, like grouping data for customers in a particular region on a specific shard.
2. **Hashed Sharding**:
    - Documents are distributed based on an MD5 hash of the shard key value.
    - Ensures an even distribution of writes across shards, beneficial for handling streams of time-series and event data.
3. **Zoned Sharding**:
    - Developers can specify rules for data placement in a sharded cluster, offering more control over how data is distributed.

Examples:

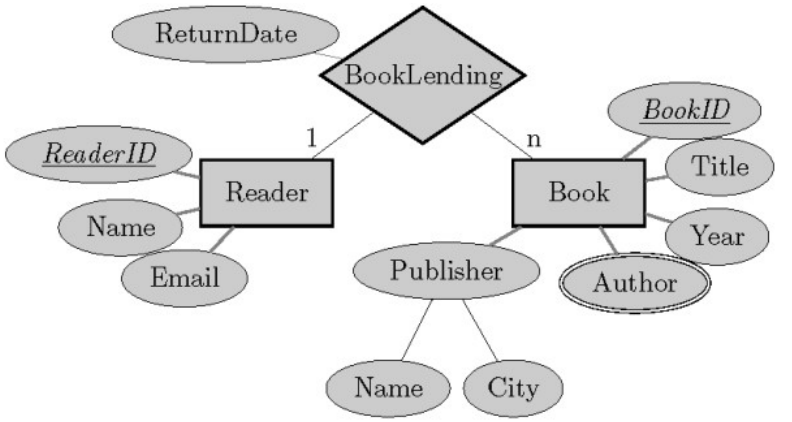
→ List of all borrowed books by a reader:

```
db.books.find({ "borrower": "reader_id_or_name" })
```

→ List of all lent books:

```
db.books.find({ "borrower": { $exists: true, $ne: "" } })
```

MongoDB usage example



Reader	ReaderID	Name
	2468	Nasr
	2512	Aboubakr

Book	BookID	Title
	1002	Introduction to DBS
	1004	Patterns of enterprise application architecture
	1006	Don Quixote

BookLending	BookID	ReaderID	ReturnDate
	1002	2468	25-10-2016
	1006	2468	27-10-2016
	1004	2512	31-10-2016

Usage of MongoDB

DTD (Document Type Definition) and XML Schema (XSD) are both used to define the structure and content rules for XML documents:

1. **DTD:**
    - Simple but limited.
    - Own syntax, not XML.
    - Specifies basic structure and content constraints.
2. **XML Schema (XSD):**
    - Complex but powerful.
    - XML standard.
    - Defines detailed structure, data types, and constraints.
    - Offers advanced validation capabilities.

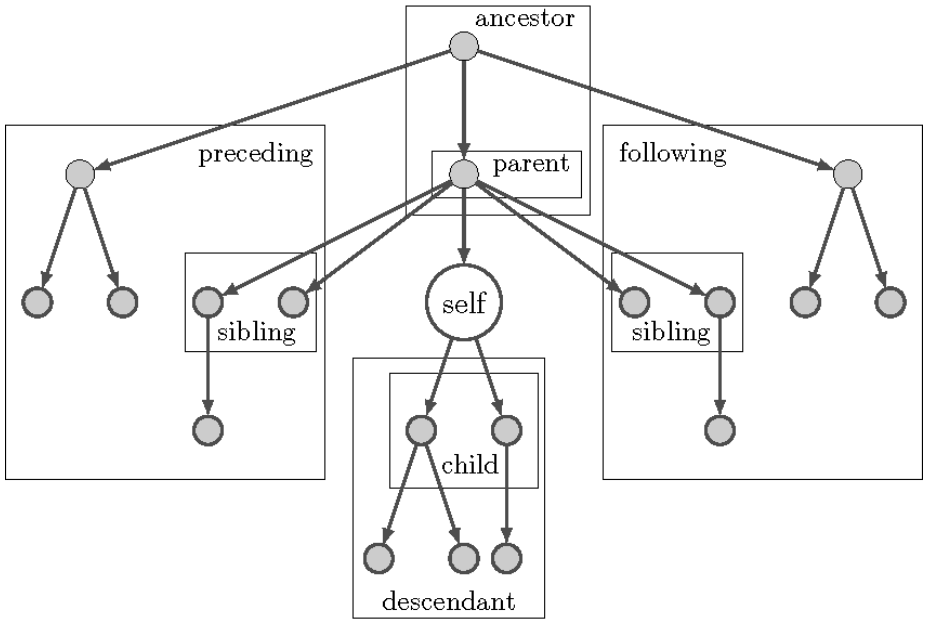
 # Why XML Schema: XML Schema (XSD) offers advantages over DTD:

1. Supports data types.
2. Allows user-defined types.
3. Enables constraints on element occurrence.
4. Supports typed references.
5. Defined in XML syntax.
6. Integrated with namespaces.
7. Provides additional features like list types and inheritance.

XML documents can be visualized as ordered trees:

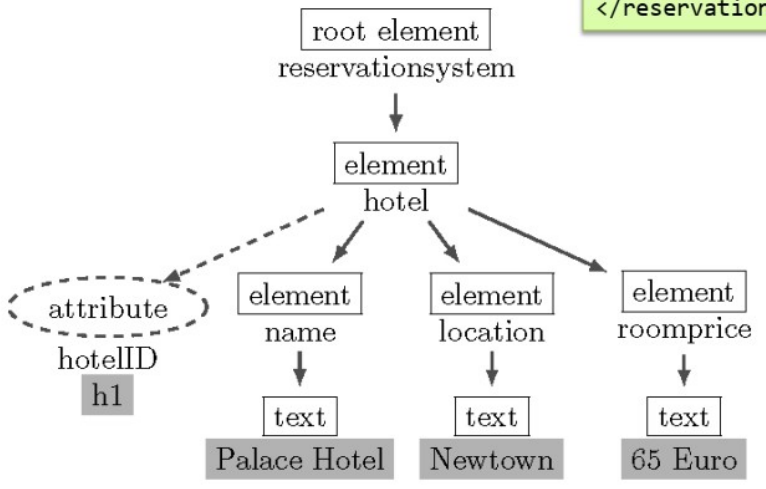
- Root element: root node
- Elements, attributes, comments, and text content are child nodes of the same parent.
- Ancestors: nodes from a parent to the root.
- Descendants: subtree starting from a node.
- Ordered tree: each node has preceding and following nodes (except root and last leaf).
- Current node during navigation is called context or self node.

This model doesn't consider IDs and IDREFs, which can create any structure.



→ Example:

```
<reservationsystem>
     <hotel hotelID="h1">
          <name>Palace Hotel</name>
          <location>Newton</location>
          <pr:roomprice>65 Euro</roomprice>
     </hotel>
</reservationsystem>
```



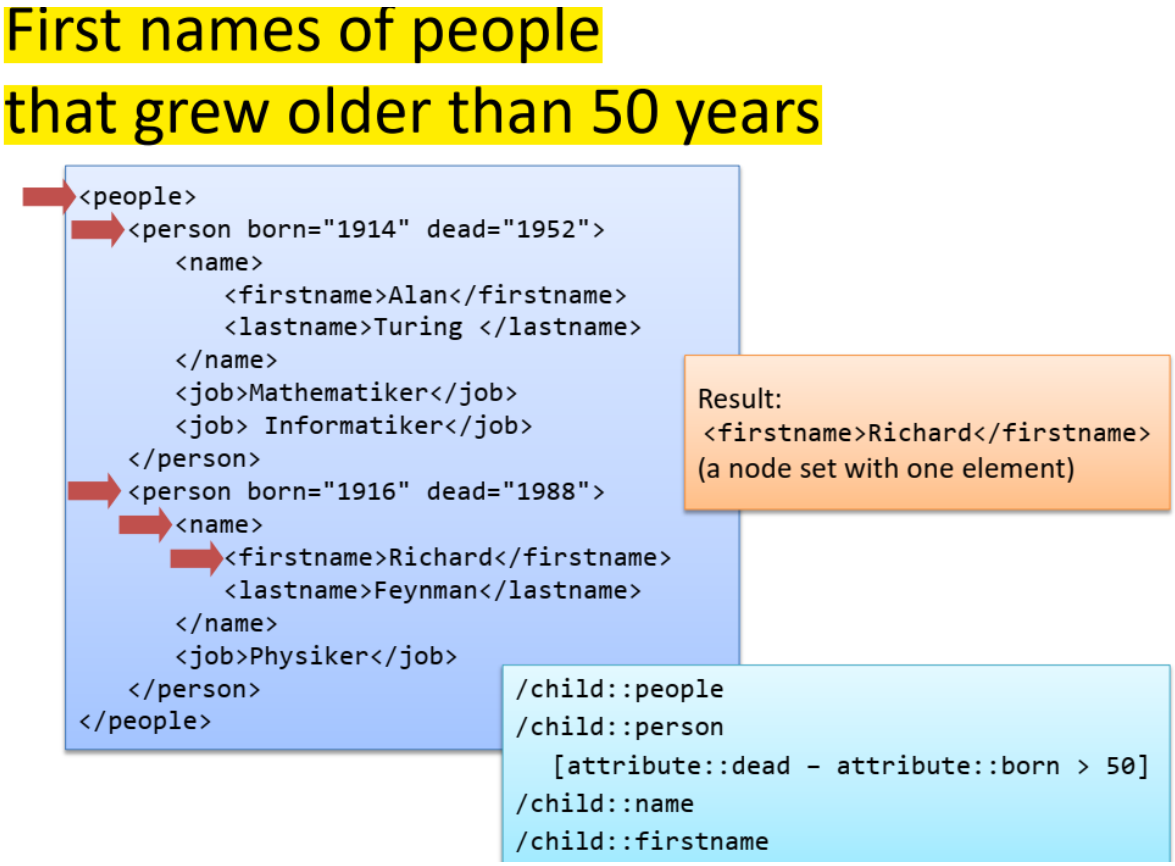
 # XML query languages include:

- **XPath**: Concise language to select parts of an XML document, resembling directory paths ("/hotel/name").
- **XQuery**: A verbose language with rich features, building on existing query languages like XPath, Quilt, and SQL.
- **XSLT**: Designed for XML transformation, converting XML to other formats like HTML or LaTeX.

 # The XML data model consists of a tree structure representing the document, with seven node types:

1. Root Node
2. Element Nodes
3. Attribute Nodes
4. Namespace Nodes
5. Processing Instruction Nodes
6. Comment Nodes
7. Text Nodes

Examples of XPath:



Names of all scientists

```
<people>
  <person born="1914" dead="1952">
    <name>
        <firstname>Alan</firstname>
        <lastname>Turing </lastname>
    </name>
    <job> Mathematiker</job>
    <job> Informatiker</job>
    <job> scientist </job>
  </person>
  <person born="1916" dead="1988">
    <name>
        <firstname>Richard</firstname>
        <lastname>Feynman</lastname>
    </name>
    <job> Physiker</job>
    <job> scientist </job>
  </person>
</people>
```

Result:

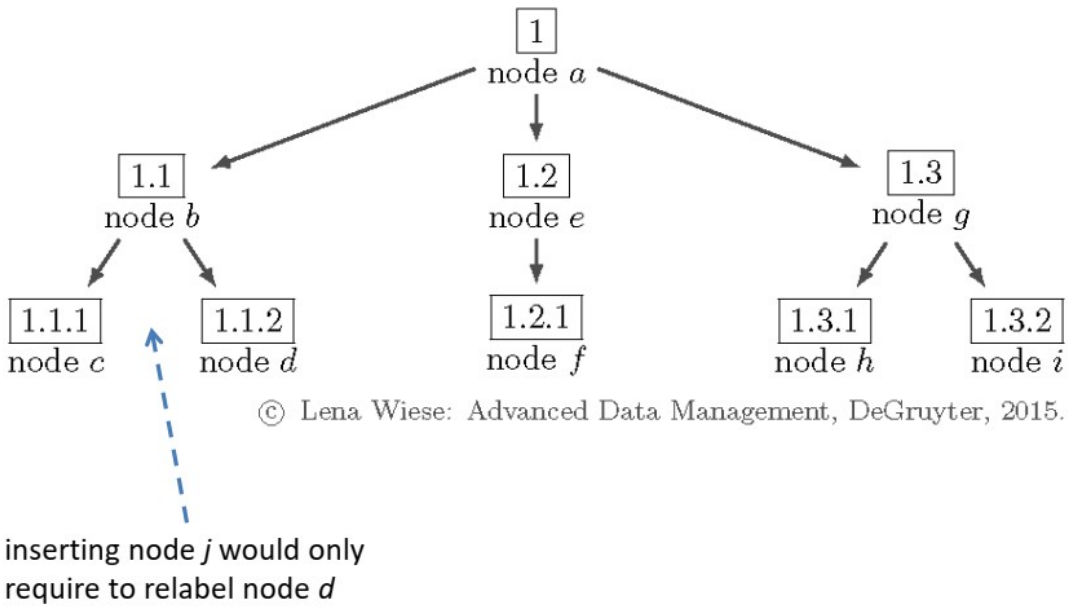
```
<name>
   <firstname>Alan</firstname>
   <lastname>Turing </lastname>
</name>
<name>
   <firstname>Richard</firstname>
   <lastname>Feynman</lastname>
</name>
```

(a node set with two elements)

```
/child::people
/child::person[child::job = "scientist"]
/child::name
```

 # Advanced XML numbering assigns unique identifiers or numbers to nodes in an XML document for efficient traversal and manipulation. Techniques include pre/post numbering and prefix numbering schemes like DeweyIDs and OrdPath, enabling operations like navigation and restructuring.

DeweyID numbering – example



inserting node *j* would only require to relabel node *d*

 # Storing XML in relational databases:

1. SQL/XML: Utilizes XML features of SQL standard.
2. Schema-based mapping: Maps XML schema to a relational schema.
3. Schemaless mapping: Maps XML document to a generic relational schema.

Choice depends on content model, DBMS features, data change rate, and schema change rate.

 # Schema-based mapping involves processing the schema information of XML documents to generate a relational schema. This process, also known as "shredding the XML," involves:

- Creating tables linked by foreign key constraints.
- Inlining sub-elements that can only occur once.
- Handling optional sub-elements by allowing NULL values in foreign keys.
- Maintaining the order of elements by storing a node ID.

XML:

```
<reservationsystem>
  <hotel hotelID='h1'>
    <name>City Residence</name>
    <location>Newtown</location>
  </hotel>
  <booking bookingID='b1' hotelbooked='h1'>
    <client>M. Mayer</client>
  </booking>
</reservationsystem>
```

Relational tables:

Reservationsystem	Id
	0

Booking	BookingID	Hotelbooked	Client	ParentID
	b1	h1	M. Mayer	0

Hotel	HotelID	Name	Location	Roomprice	ParentID
	h1	City Residence	Newtown	NULL	0

 # Schemaless mapping:

1. Utilizes a fixed generic database schema.
2. Creates a tuple for each XML node.
3. Stores node information like node name and data in separate columns.
4. Resembles the DOM (document object model) structure.
5. XML queries are translated into SQL queries, potentially involving self-joins of generic tables.

Schemaless mapping: Example

XML:

```
<reservationsystem>
  <hotel hotelID='h1'>
    <name>City Residence</name>
    <location>Newtown</location>
  </hotel>
  <booking bookingID='b1' hotelbooked='h1'>
    <client>M. Mayer</client>
  </booking>
</reservationsystem>
```

Generic relational table:

xmlData	nodeID	nodeType	nodeName	nodeData	parentID
	0	root	resevationsystem	NULL	NULL
	1	element	hotel	NULL	0
	2	attribute	hotelID	h1	1
	3	element	name	NULL	1
	4	text	NULL	Palace Hotel	3
	5	element	location	NULL	1
	6	text	NULL	Newtown	5
	7	element	roomprice	NULL	1
	8	text	NULL	65 Euro	7

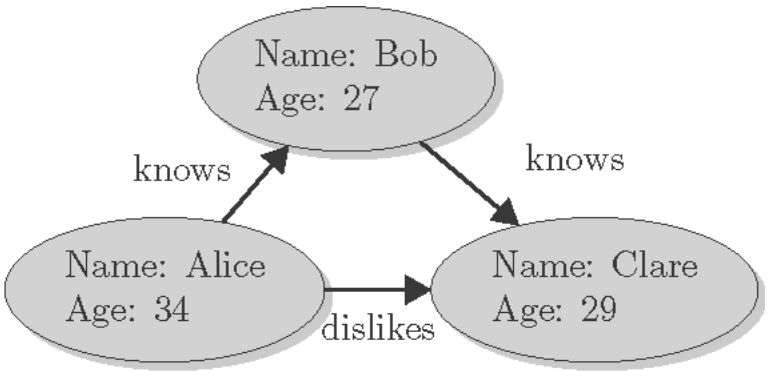
Graph Database

 # Why Graph Databases?

• A graph is a natural way to model arbitrary connections between (data) objects.

 # Goals of Graph Databases:

1. Flexibility: Enable natural modeling of data and relationships.
2. Agility: Support schema-less data management for quick adaptation.
3. Performance: Provide efficient access to connected data structures.
4. Non-Goal: Scalability (scale out), as partitioning a graph is challenging.



 # Graph Theory: G = (V, E)
– V: a set of vertices
– E: a set of edges (pairs of vertices that are connected)

• Edge types:
– directed (denoted as arrow):
from first vertices to second vertices
– un-directed (just a line)

 # Graph Traversal:

- Full graph traversal visits each node.
- Partial graph traversal visits only certain nodes, such as those within a specific path length or with particular properties.

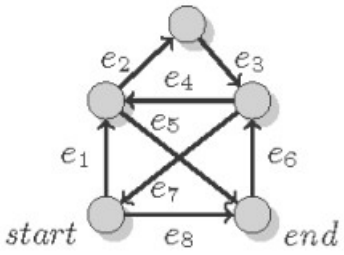
Traversal strategies include:

- Depth-first: Follows edges to adjacent nodes before exploring other edges from the starting node.
- Breadth-first: Visits all adjacent nodes before moving to nodes at the next level.

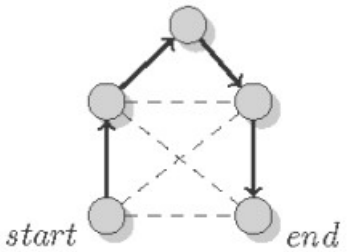
 # Graph problems include:

- **Eulerian Path**: A path that visits each edge exactly once, where the starting and ending nodes may differ, and nodes can be visited multiple times.
- **Eulerian Circle**: An Eulerian path that ends at the starting node.
- **Hamiltonian Path**: A path that visits each node exactly once, without necessarily traversing all edges.
- **Spanning Tree**: A subset of edges forming a tree, starting from a root node and visiting each node of the tree.

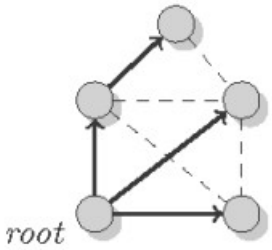
Eulerian Path



Hamiltonian Path



Spanning Tree



 # All the Graph types:

1. Simple Undirected Graph:
    - Consists of vertices (nodes) connected by edges, where the edges have no direction.
    - Example: A social network where people are represented by vertices and friendships between them are represented by edges. Each edge simply denotes a connection between two people, without specifying a direction.
2. Simple Directed Graph:
    - Similar to the undirected graph, but edges have a direction, indicating a one-way relationship between vertices.
    - Example: A network of roads where intersections are represented by vertices and roads connecting them are represented by directed edges. Each edge indicates the direction of travel between two intersections.

3. Undirected Multigraph:

- Allows multiple edges between the same pair of vertices, but no self-loops (an edge connecting a vertex to itself).
- Example: A transportation system where multiple train tracks connect the same pair of cities, allowing for different routes between them.

4. Directed Multigraph:

- Similar to the undirected multigraph, but edges have direction, and multiple directed edges between the same pair of vertices are allowed.
- Example: An airline route network where multiple flights operate between the same pair of airports, but they may have different departure times or routes.

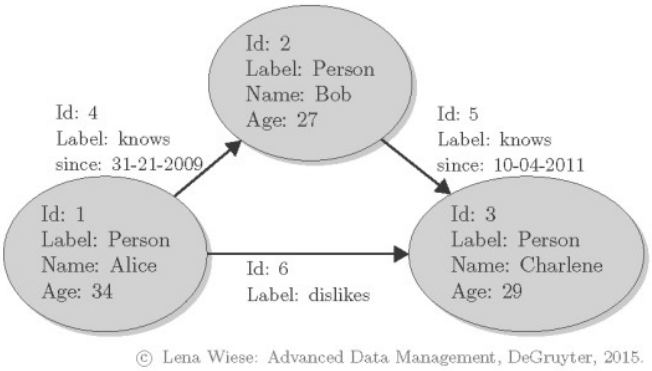
5. Weighted Graph:

- Assigns a weight or cost to each edge, indicating the "cost" of traversing that edge.
- Example: A map where vertices represent cities and edges represent roads, with each edge having a weight corresponding to the distance between the cities it connects.

6. Property Graph:

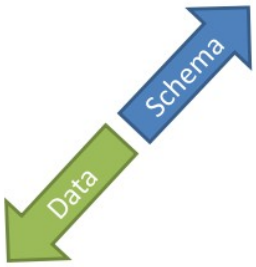
- Extends the concept of a graph to include properties or attributes associated with both vertices and edges.
- Example: A knowledge graph representing relationships between entities, where vertices represent entities (e.g., people, places) and edges represent relationships (e.g., "likes," "works at"), with additional properties such as timestamps or labels attached to vertices and edges.

Property graph – example



Graph definition:

$G = (V, E, L_V, L_E, ID)$
$L_V = \{Person\}$
$t_{Person} = \{Person, A_{Person}\}$
$A_{Person} = \{(Name, \text{String}), (Age, \text{Integer})\}$
$L_E = \{knows, dislikes\}$
$t_{knows} = \{knows, A_{knows}, \{Person\}, \{Person\}\}$
$t_{dislikes} = \{dislikes, \emptyset, \{Person\}, \{Person\}\}$
$A_{knows} = \{(since, \text{Date})\}$
$A_{dislikes} = \emptyset$



Nodes and Edges:

$V = \{v_1, v_2, v_3\}$
$E = \{e_1, e_2, e_3\}$
$ID = \{1,2,3,4,5,6\}$

$v_1 = \{1, Person, \{Name: \text{Alice}, Age: 34\}\}$
$v_2 = \{2, Person, \{Name: \text{Bob}, Age: 27\}\}$
$v_3 = \{3, Person, \{Name: \text{Charlene}, Age: 29\}\}$
$e_1 = \{4, knows, \{since: 31-21-2009\}, 1,2\}$
$e_2 = \{5, knows, \{since: 10-04-2011\}, 2,3\}$
$e_3 = \{6, dislikes, \emptyset, 1,3\}$

 # Graph Data Structures Overview:

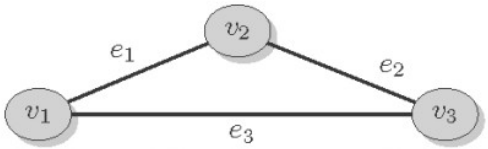
1. Edge List:

- Represents a graph as a list of edges, each edge containing the pair of vertices it connects.
- Simplest representation but not memory-efficient for large graphs.
- Example: `[(A, B), (A, C), (B, C), (B, D)]`

2. Adjacency Matrix:

- Represents a graph as a 2D array where the presence of an edge between vertices is indicated by a non-zero value.
- Efficient for dense graphs but memory-intensive for sparse graphs.

- Example: Simple undirected graph:-



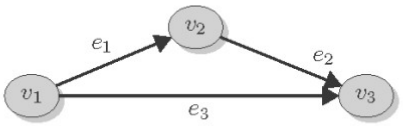
	v_1	v_2	v_3
v_1	0	1	1
v_2	1	0	1
v_3	1	1	0

or

	v_1	v_2	v_3
v_1	0		
v_2	1	0	
v_3	1	1	0

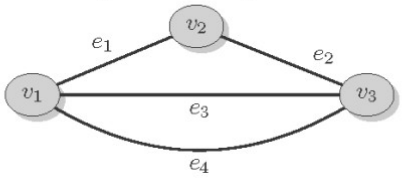
- For simple directed graph, the matrix is asymmetric:

	v_1	v_2	v_3
v_1	0	0	0
v_2	1	0	0
v_3	1	1	0



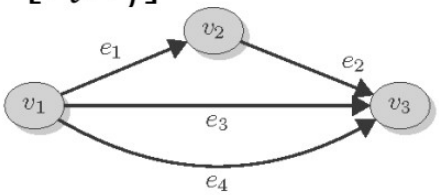
- For a undirected multigraph, if k edges $\{v_i, v_j\}$ exist, write k in the matrix cells $[v_i, v_j]$ and $[v_j, v_i]$:

	v_1	v_2	v_3
v_1	0	1	2
v_2	1	0	1
v_3	2	1	0



- For an directed multigraph, if k edges (v_i, v_j) exist, write k in the matrix cells $[v_i, v_j]$:

	v_1	v_2	v_3
v_1	0	0	0
v_2	1	0	0
v_3	2	1	0



3. Incidence Matrix:

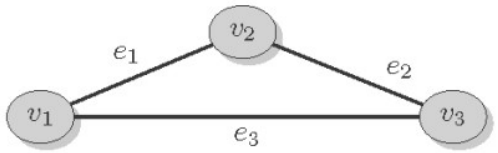
- Represents a graph as a 2D array where rows represent vertices and columns represent edges, with entries indicating whether a vertex is incident to an edge.
- Suitable for edge-based operations but not commonly used due to its size.

- Example:

Simple undirected graphs:

- write a 1 in $[i, j]$ if edge e_i is connected (incident) to v_j
- if e_i is a loop $\{v_i, v_i\}$, write a 2 in $[i, j]$

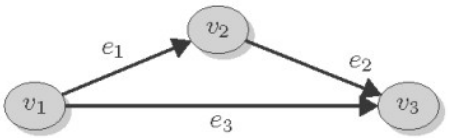
	e_1	e_2	e_3
v_1	1	0	1
v_2	1	1	0
v_3	0	1	1



Simple directed graph

- Write -1 for the source node v_i and 1 for the target node v_j
- if e_i is a loop (v_i, v_i), write a 2 in $[i, j]$

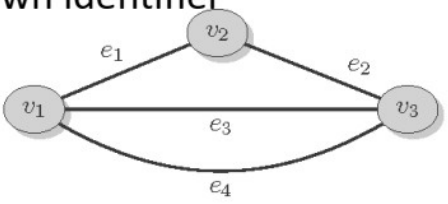
	e_1	e_2	e_3
v_1	-1	0	-1
v_2	1	-1	0
v_3	0	1	1



Undirected multigraph

- Same procedure as undirected simple graph, because each edge has ist own identifier

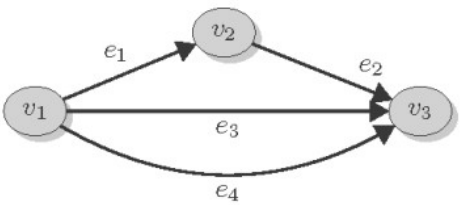
	e_1	e_2	e_3	e_4
v_1	1	0	1	1
v_2	1	1	0	0
v_3	0	1	1	1



Directed multigraph:

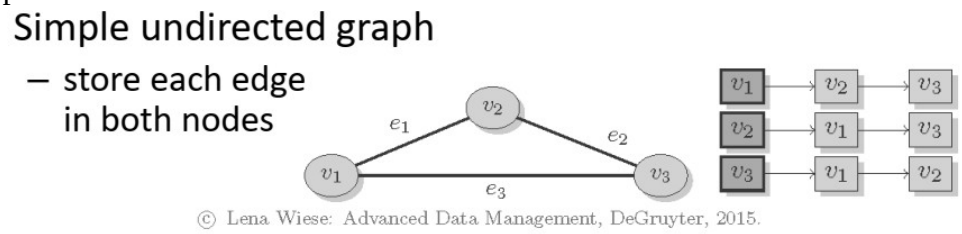
- Same procedure as for directed simple graph, because each edge has ist own identifier

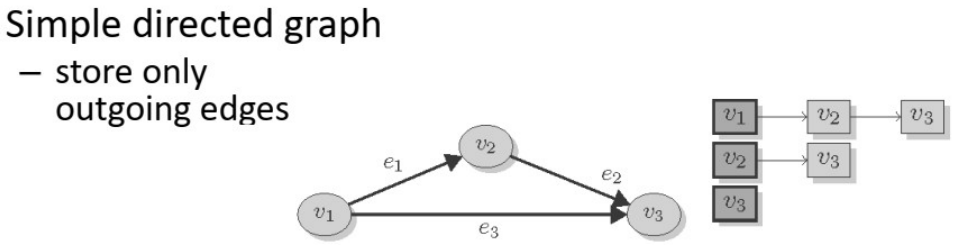
	e_1	e_2	e_3	e_4
v_1	-1	0	-1	-1
v_2	1	-1	0	0
v_3	0	1	1	1



4. Adjacency List:

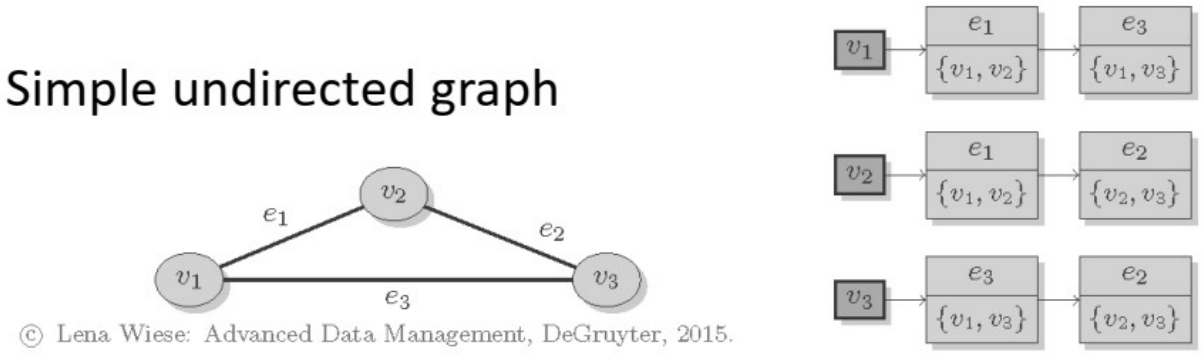
- Represents a graph as a collection of lists, where each vertex has a list of adjacent vertices.
- More memory-efficient for sparse graphs but slower for certain operations.
- Example:

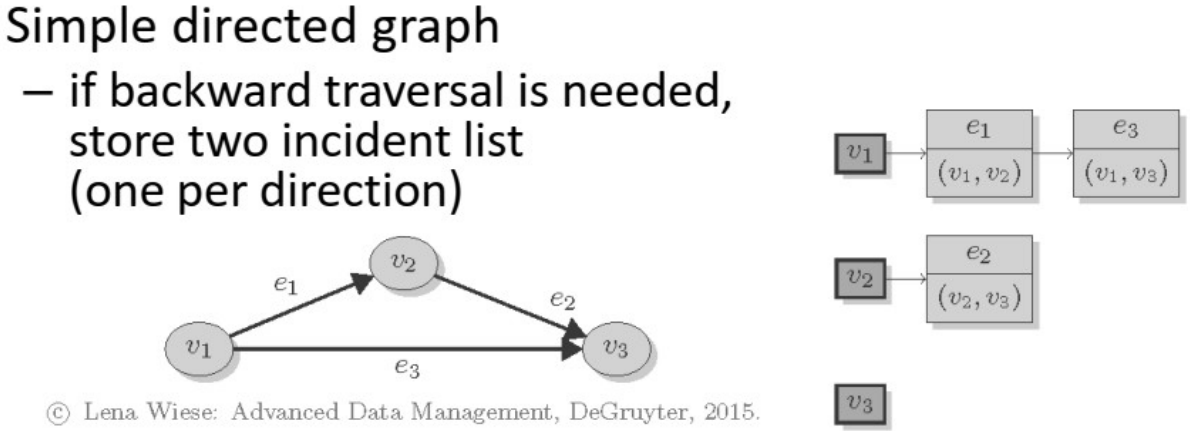




5. Incidence List:

- Represents a graph as a list of edges, where each edge contains references to its incident vertices.
- Suitable for situations where edge-based operations are more common.
- Example:





6. Property Graph in Relational Tables:

- Represents a graph using relational database tables, with separate tables for vertices, edges, and properties.
- Example:
  - Vertices Table: (id, label, property1, property2, ...)

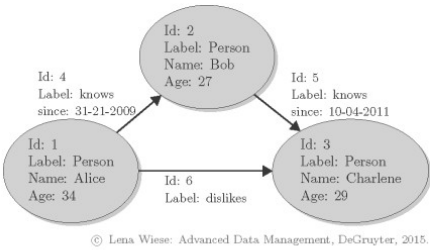
- Edges Table: (id, source_vertex_id, target_vertex_id, label, property1, property2, …)

One attribute table

Nodes	NodeID	NodeLabel
	1	Person
	2	Person
	3	Person

Edges	EdgeID	EdgeLabel	Source	Target
	4	knows	1	2
	5	knows	2	3
	6	dislikes	1	3

Attributes	ID	PropertyKey	PropertyValue
	1	Name	Alice
	1	Age	34
	2	Name	Bob
	2	Age	27
	3	Name	Charlene
	3	Age	29
	4	since	31-21-2009
	5	since	10-04-2011



☺ Only one table for all attributes

☹ Attributes cannot be typed

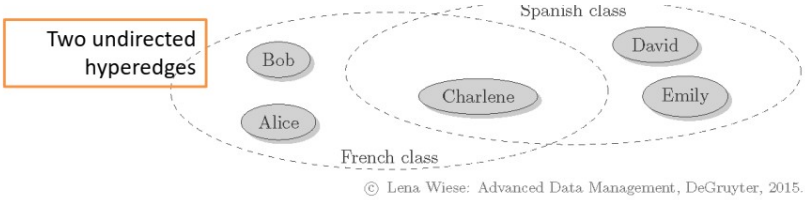
☹ Consistency needs to be checked in application, e.g.:

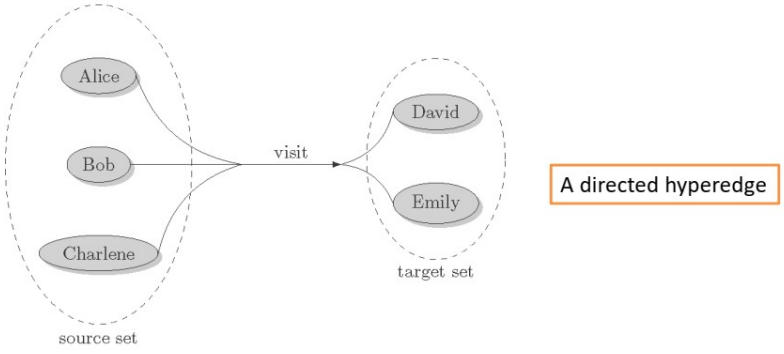
- is an age an integer?
- does the "dislikes" really have no attributes?

 # Advanced Graph Models:

1. Hypergraph:

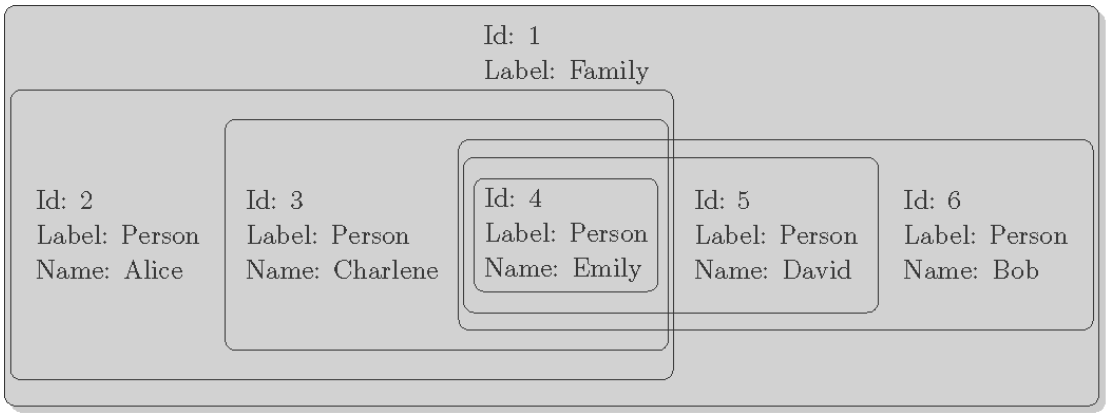
- Generalization of graphs where edges can connect more than two nodes, known as hyperedges.
- Hyperedges can be n-ary, allowing for complex relationships between multiple nodes.
- Example: In a social network, a hyperedge could represent a group of users who all attended the same event.





2. Nested Graph:

- Allows nodes to be organized into nested structures or sets, known as hypernodes.
- Hypernodes can contain other hypernodes or individual nodes, forming hierarchical relationships.
- Example: In a file system, directories can contain files and other directories, creating a nested graph structure.



 # Neo4J:

- Graph database utilizing the property graph data model.
- Relationships between nodes are called "relationships," allowing for flexible and rich connections.
- Supports transactions with ACID (Atomicity, Consistency, Isolation, Durability) properties, ensuring data integrity.
- Cypher is the query language used in Neo4J, designed specifically for graph database queries, offering expressive and intuitive syntax for traversing and querying graph structures.

SQL Statement

```
SELECT name FROM Person
LEFT JOIN Person_Department
  ON Person.Id = Person_Department.PersonId
LEFT JOIN Department
  ON Department.Id = Person_Department.DepartmentId
WHERE Department.name = "IT Department"
```

Cypher Statement

```
MATCH (p:Person)<-[:EMPLOYEE]-(d:Department)
WHERE d.name = "IT Department"
RETURN p.name
```

Graph DB vs Relational DB:

Graph DB	Relational DB
• Good for storing connected data set (of different structure)	• Good for storing similar data (relations) with some relationships
• Support multi-hop queries with unknown number of hops	• Support complex queries, but only known or limited number of „hops“ (=Joins)
• Flexible schema	• Rather fixed schema

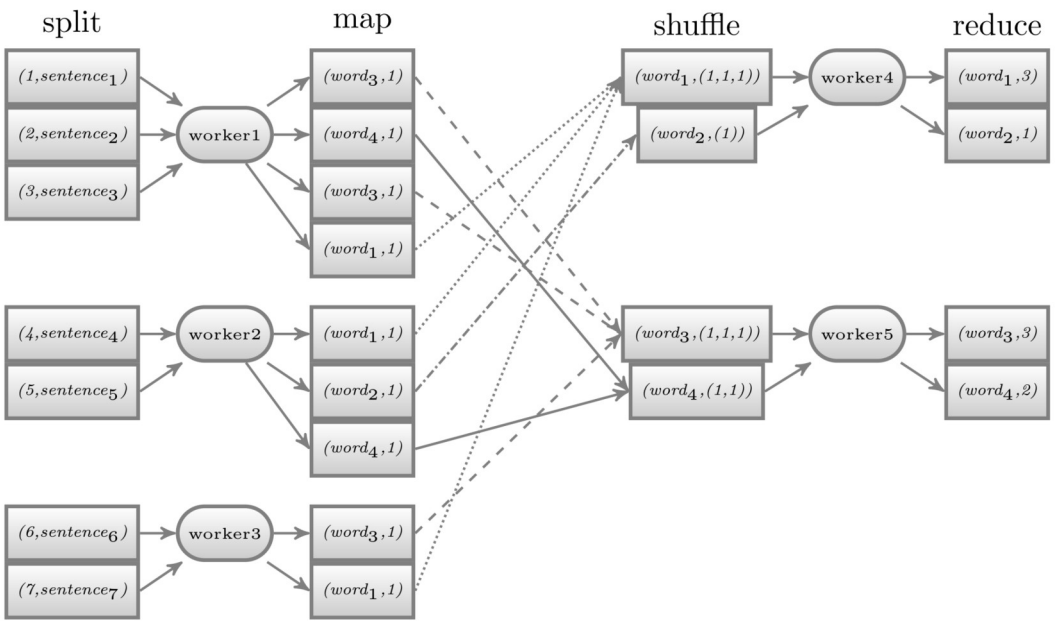
More Data Models

 # Key-value storage systems are based on a straightforward principle: associating unique keys with data values. They support basic operations like storing, retrieving, and deleting data using keys. Key-value stores offer:

1. **Easy distribution**: They can be distributed across multiple servers, making them suitable for data-heavy applications.
2. **Simplicity and speed**: Retrieving data is fast but requires knowing the specific key, as key-value stores lack advanced search capabilities.
3. **Integration with other systems**: While efficient for storage and retrieval, additional frameworks or systems may be necessary for complex data processing tasks.

MapReduce is a distributed programming model designed for processing large datasets in parallel.

- Originally developed by Google, it simplifies distributed data processing, making it suitable for big analytics tasks.
- It consists of basic steps: Split, Map, Shuffle, and Reduce.
- Input data is split and processed by worker nodes, where the results are mapped into key-value pairs.
- The results are then shuffled based on keys and reduced to produce the final output.

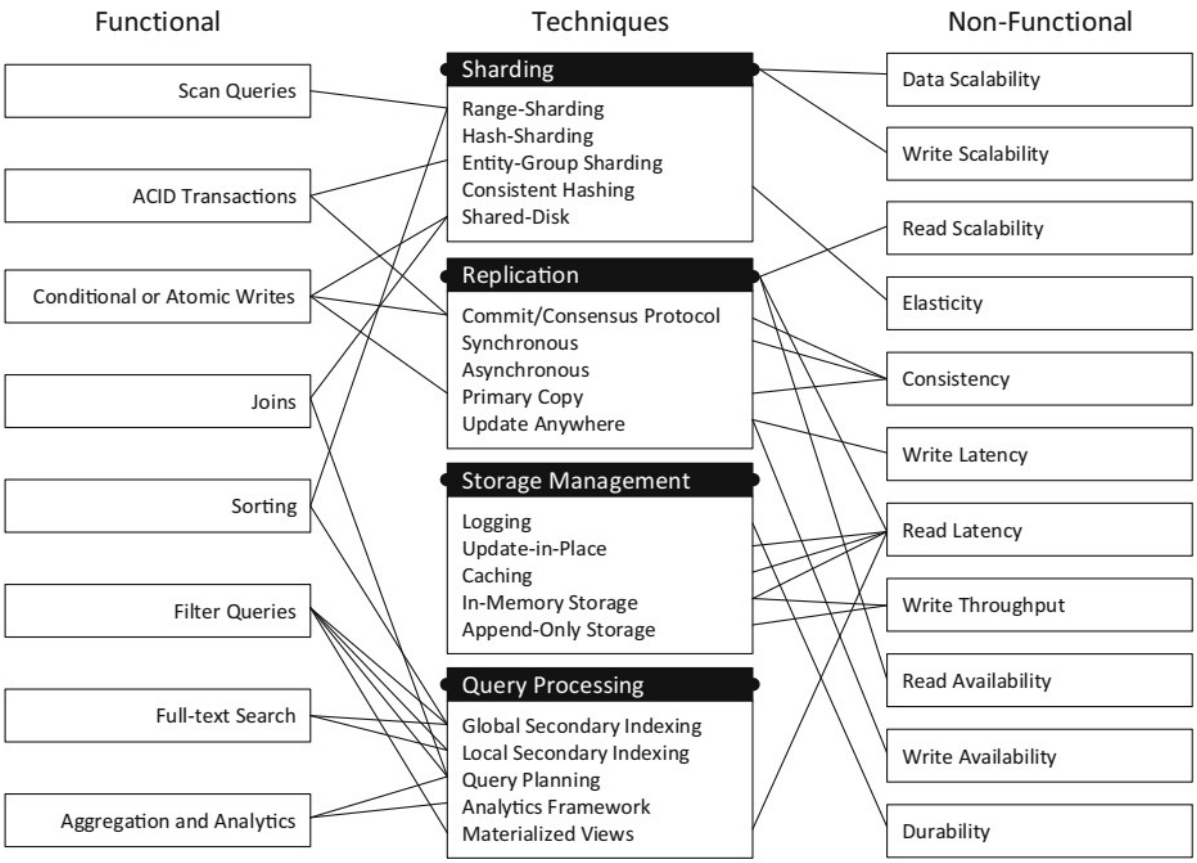


© Lena Wiese: Advanced Data Management, DeGruyter, 2015.

 # Features and Optimization of Map/Reduce:

- **Parallelization:** Map and Reduce tasks can run on multiple nodes concurrently, coordinated by a master process.
- **Partitioning:** Reduce tasks are distributed based on key hashing, ensuring balanced workload.
- **Combination:** Combine tasks can locally aggregate data before sending to reduce tasks, reducing intermediate results.
- **Data Locality:** Tasks can be assigned to nodes already holding relevant data, minimizing data transfer.
- **Incremental Map-Reduce:** Steps can be interleaved to produce results incrementally, but feasibility depends on the application.

 # Paper:


Functional
Techniques
Non-Functional
Scan Queries
ACID Transactions
Conditional or Atomic Writes
Joins
Sorting
Filter Queries
Full-text Search
Aggregation and Analytics
Sharding
Range-Sharding
Hash-Sharding
Entity-Group Sharding
Consistent Hashing
Shared-Disk
Replication
Commit/Consensus Protocol
Synchronous
Asynchronous
Primary Copy
Update Anywhere
Storage Management
Logging
Update-in-Place
Caching
In-Memory Storage
Append-Only Storage
Query Processing
Global Secondary Indexing
Local Secondary Indexing
Query Planning
Analytics Framework
Materialized Views
Data Scalability
Write Scalability
Read Scalability
Elasticity
Consistency
Write Latency
Read Latency
Write Throughput
Read Availability
Write Availability
Durability