

# Advanced Topics of Software Engineering (ASE)

## Chapter 3. Software architectures and their trade-offs

Prof. Dr. Florian Matthes, Prof. Dr. Alexander Pretschner

Chair of Software Engineering for Business Information Systems (sebis)  
Faculty of Informatics  
Technische Universität München  
[www.matthes.in.tum.de](http://www.matthes.in.tum.de)

## **3.1. Introduction to distributed systems and middleware**

3.2. Database-centric architectures

3.3. Message-oriented architectures

3.4. Object-oriented architectures

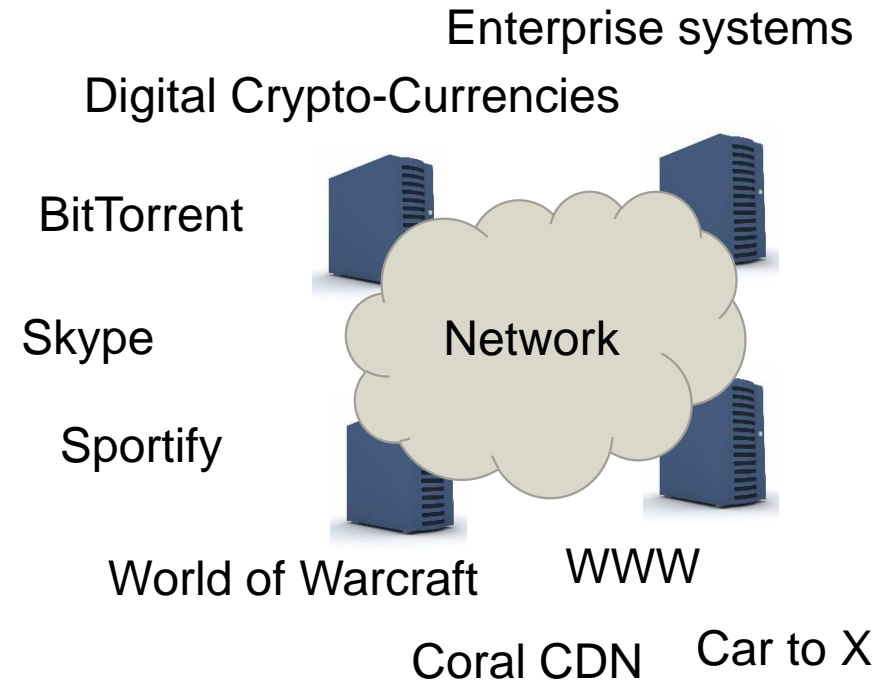
3.5. Component-based architectures

3.6. Service-oriented architectures

3.7. Blockchain-based systems

Distributed system: Physically disjoint compute resources, interconnected by a network

- A distributed system is one in which **hardware** or **software components** located at **networked computers communicate** and **coordinate** their actions only by **passing messages**.  
*By Coulouriset al.*
- A distributed system is a **collection of independent computers** that appears to its users as **a single coherent system**. *By Tanenbaum & van Steen.*



A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

[Leslie Lamport (1987)]

# Characteristics of a distributed system

- Reliability
- Availability
- Heterogeneity (variety and differences in networks, hardware, operating systems, programming languages ...)
- Openness (concerned with extensions)
- Security
- Scalability (accommodate more users, respond faster)
- Fault-tolerance and failure handling
  - Hardware, software, and networks may fail
  - Fault tolerance through recovery, redundancy ...
- Concurrency
- Transparency (access, location, concurrency, replication)
- Predictable performance

Probability of a system to **perform** its **required functions** under **stated conditions** for a **specified period of time**.

- To run continuously without failure (that is: correctly)
- Expressed as mean time between failures (MTBF), failure rate

Proportion of the time a system is in a **functioning state**, i.e., can be used.

- **Ratio** of time usable over entire time
  - Informally,  $uptime / (uptime + downtime)$
  - System that **can be used** 100 hrs. out of 168 hrs. has availability of 100/168
  - System could be up, but not usable (outage!)
- Specified as decimal or percentage
- Five nines is 0.99999 or 99.999% available
- Could be represented as  $(1 - Unavailability)$

# Characteristics of a distributed system

Measure for availability (Nines / Class of 9)

# Nines	Unav.	Avail. (%)	Downtime per			
			year	month	week	day
1 x 9	$10^{-1}$	90	36.5 d	3d	16.8 h	2.4 h
4 x 9	$10^{-4}$	99.99	52.56 min	4.32 min	60.48 s	8.64 s
5 x 9	$10^{-5}$	99.999	5.256 min	25.92 s	6.048 s	864 ms
6 x 9	$10^{-6}$	99.9999	31.536 s	2.592 s	604.8 ms	86.4 ms
9 x 9	$10^{-9}$	99.99999999	31.536 ms	2.592 ms	604.8 $\mu$ s	86.4 $\mu$ s

- Frequently used for telecommunication systems
- Favorite marketing term
- Does not capture impact or cost of downtime

Availability is not equal to reliability

- Reliable system has high availability but an available system may or may not be reliable.
- Server may never be down but sometimes cause incorrect results, e.g., as a result of ill-functioning synchronization.



Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource is replicated
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource

Middleware comprises **services** and **abstractions** that facilitate the design, development, and deployment of **distributed applications** in heterogeneous, networked environments.

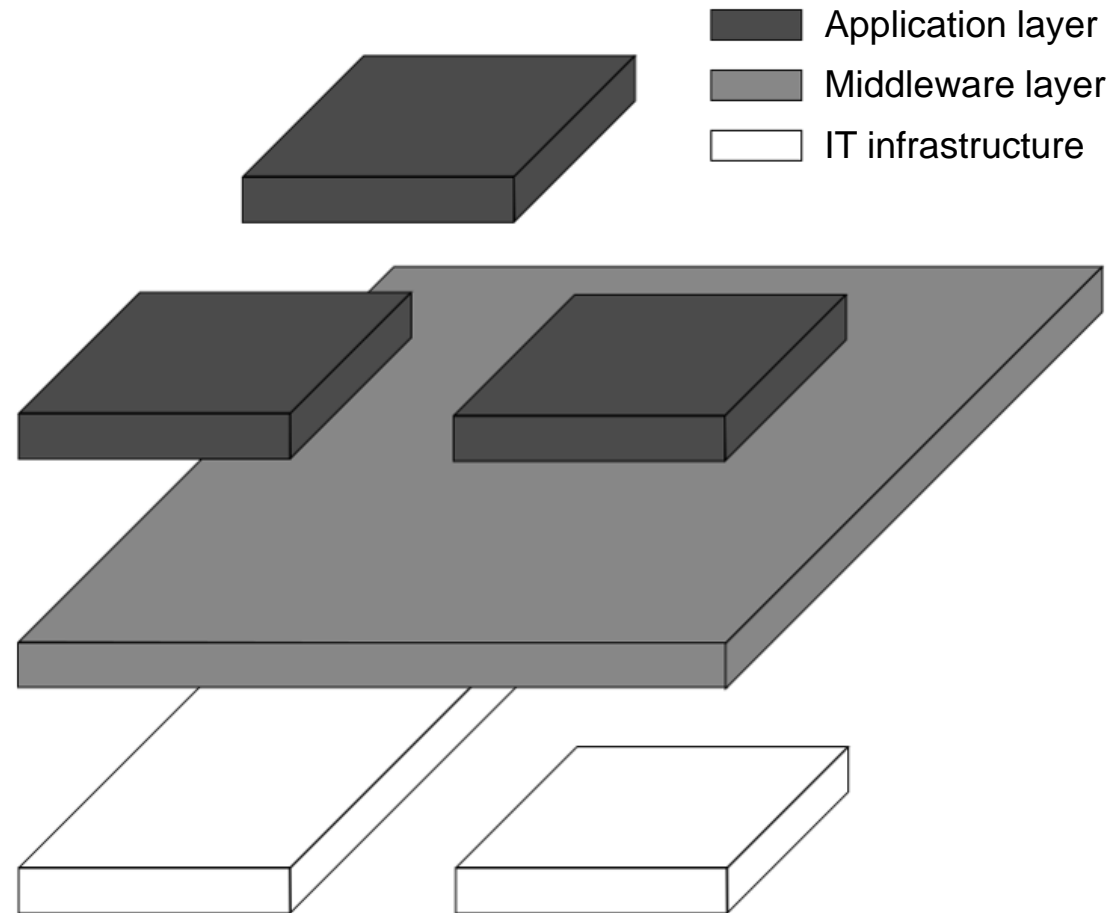
**Example abstractions:** Remote invocation, messaging, publish/subscribe, TP monitor, locking service, etc.

**Examples:** DCE, CORBA, RMI, JMS, Web services, etc.

## Middleware (2)

- Captures **common functionalities**
  - Message passing, remote invocation
  - Message queuing, publish/subscribe
  - Transaction processing
  - Naming, directory, security provisions
  - Fault-tolerance, consistent views
  - Replication, availability
- Deals with **interoperability**
- Deals with **system integration**

# Middleware stack



## 3.1. Introduction to distributed systems and middleware

## **3.2. Database-centric architectures**

### 3.2.1. Data warehousing and business intelligence

### 3.2.2. Big data architectures

## 3.3. Message-oriented architectures

## 3.4. Object-oriented architectures

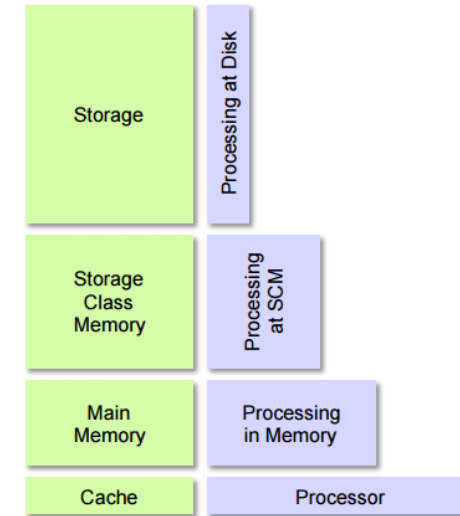
## 3.5. Component-based architectures

## 3.6. Service-oriented architectures

## 3.7. Blockchain-based systems

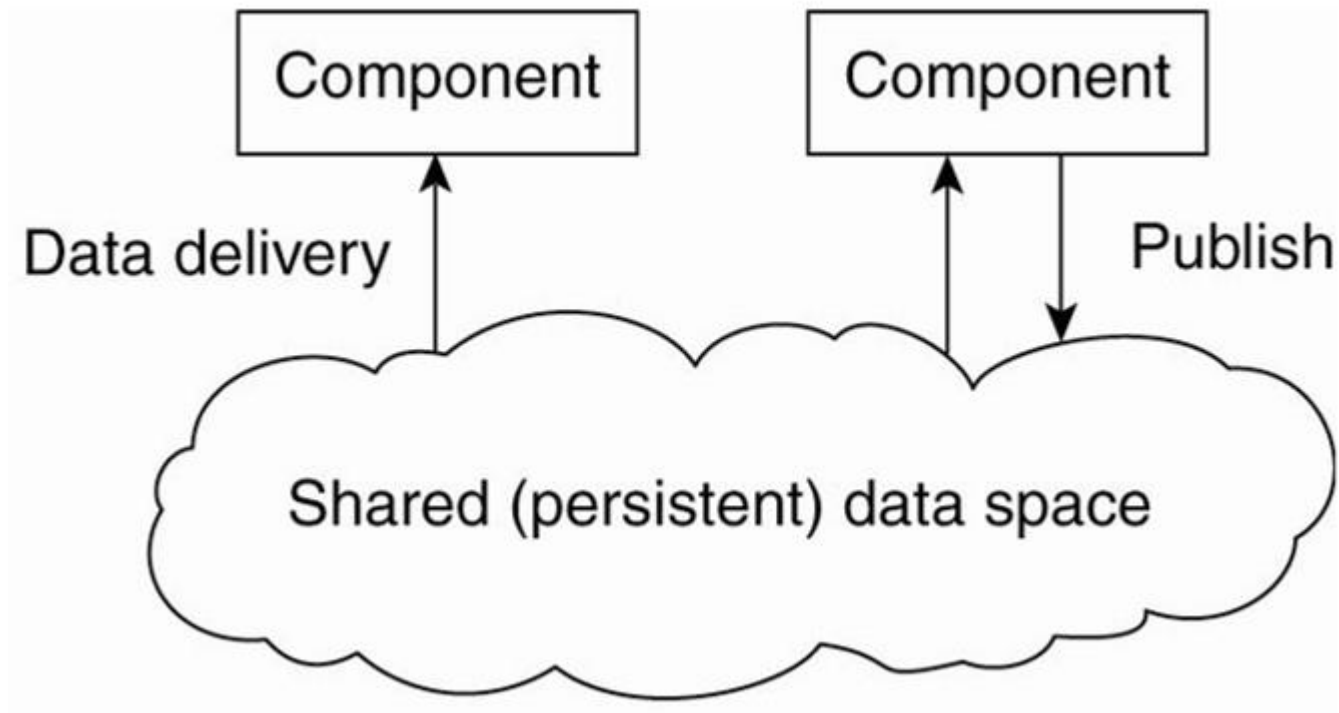
# Database-centric architectures

- Main purpose: *data access and update*
  - Databases play a crucial role



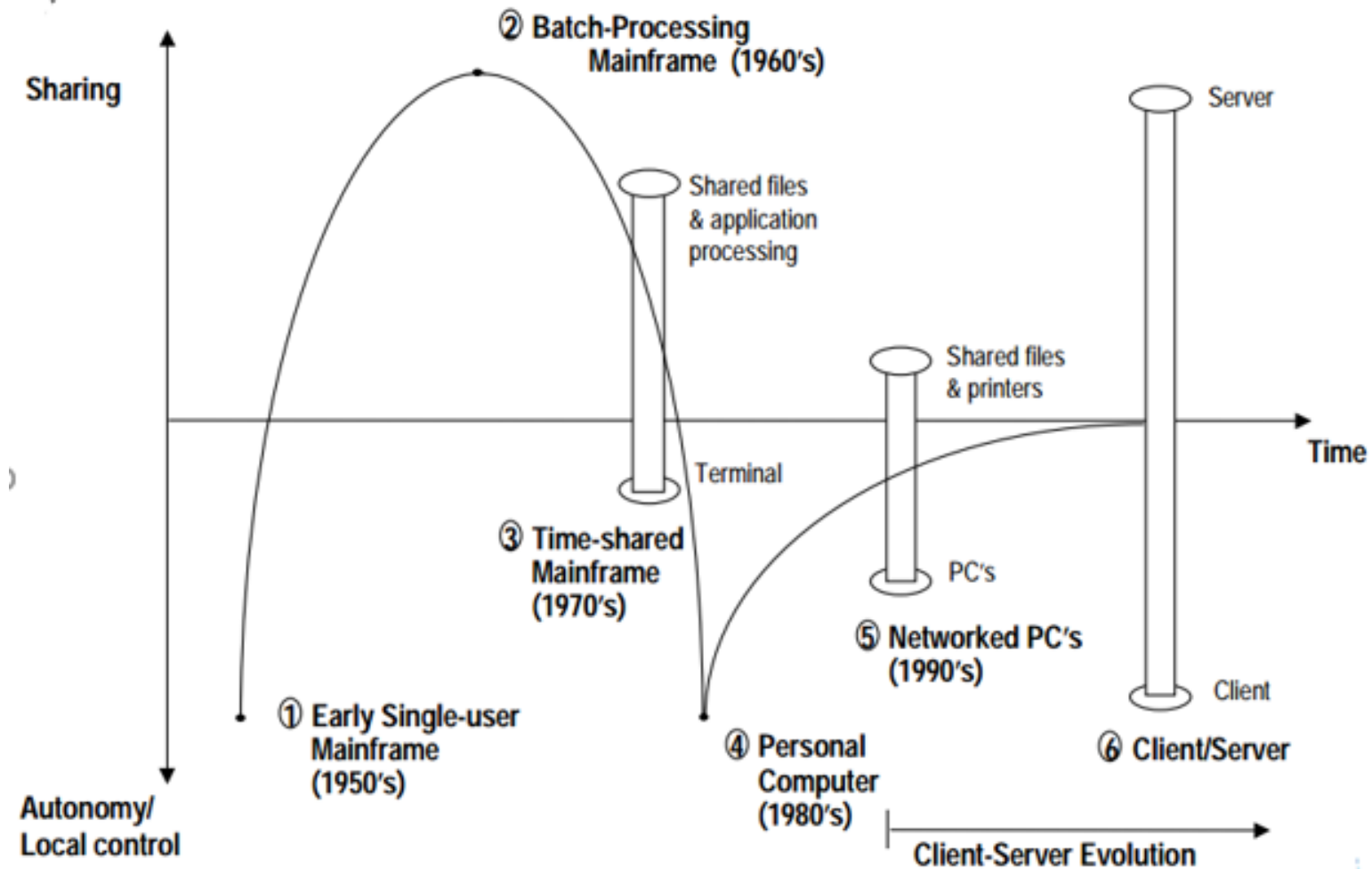
- *Processes interact by reading and modifying data in some shared repository* (active or passive)
  - Traditional data base (passive): Responds to requests
  - Blackboard system (active): Clients solve problems collaboratively; system updates clients when information changes.
- Favors *stored procedures* that run on database servers, as opposed to greater reliance on logic running in middle-tier application servers in a multi-tier architecture.

# Shared data-space architectural style



[“Distributed systems: principles and paradigms.” Tanenbaum and Steen (2007)]

# The client-server evolution

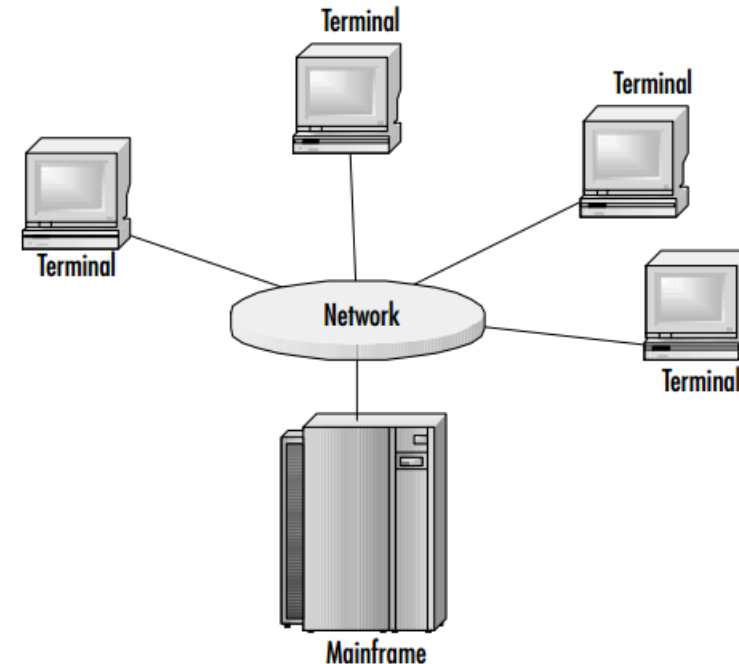


[“Evolving IT architectures: from mainframes to client-server to network computing.” S. Madnick (1998)]



## A typical mainframe model

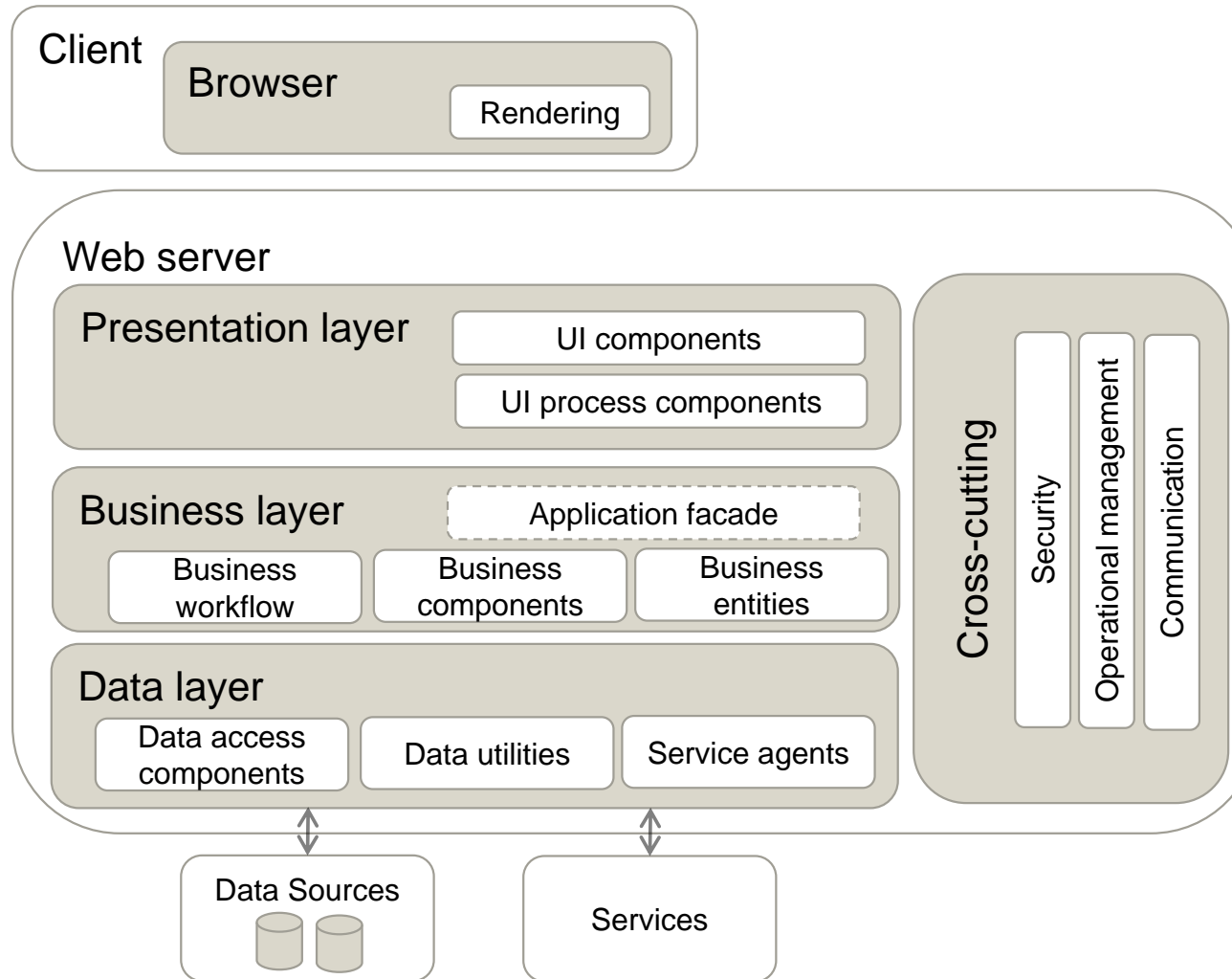
- The mainframe model uses the mainframe as a central repository for information as well as processing for every application
- Terminals enable input/output into the applications
- All administration takes place on the mainframe itself



- Hardware maintenance cost reduction
- Single point of administration
- One type of administrative skill set
- Simple architecture and low bandwidth requirements

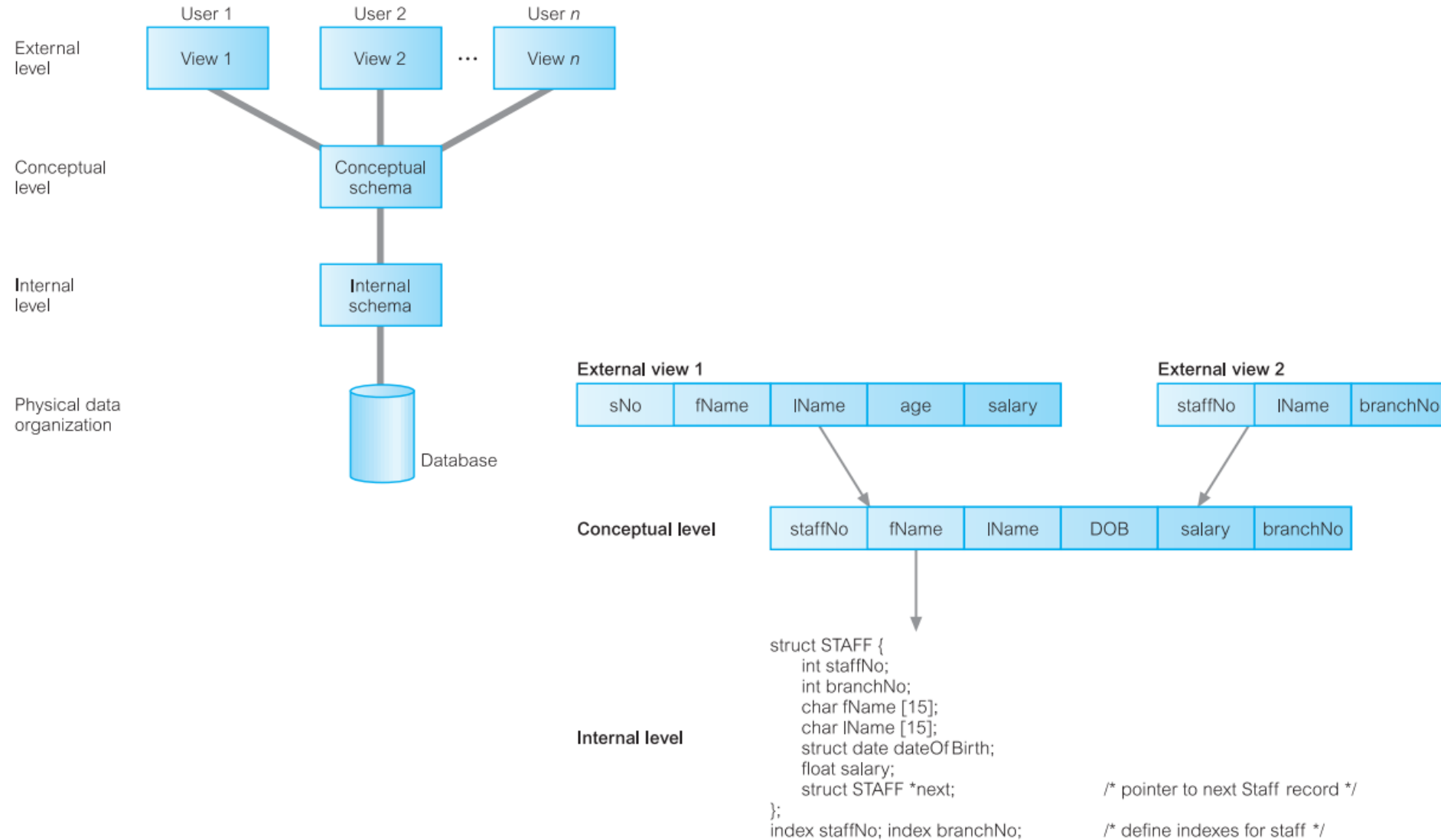
- Single point of failure
- Character-based applications
  - Mainframe applications are generally textual
- Bottlenecks due to time-sharing systems

# A typical three-layered client/server architecture



- Reduced hardware costs
- No single point of failure
- Flexibility
- Scalable architecture

- Heightened administrative costs
- Increased security risks
- Lack of centralized backup



### Base relation

A named relation corresponding to an entity in the conceptual schema whose tuples are physically stored in the database.

### View

- The dynamic result of one or more relational operations operating on the base relations to produce another relation
- A view is a **virtual relation** that does not actually exist in the database but is produced upon request, at the time of request

- Views are dynamic, i.e. changes made to base relations that affect view attributes are immediately reflected in the view
- Example

```
CREATE VIEW profNumStudents AS
SELECT pNumber, count(*) AS CNT
FROM Student
GROUP BY pNumber;
```

## Purpose of views

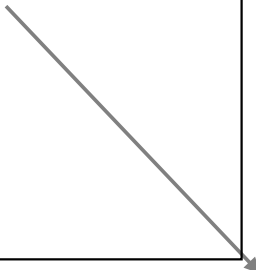
- Provides a powerful and flexible security mechanism by hiding parts of database from certain users
- Permits users to access data in a customized way, so that same data can be seen by different users in different ways, at same time
- It can simplify complex operations on base relations
- Complex queries can be reused
- Logical data independence (the base table may change but the view could remain the same)

- Data independence
- Improved security
- Reduced complexity
- Convenience
- Customization
- Data integrity

- Update restriction
- Structure restriction
- Performance

- **Subprograms** are named PL/SQL blocks that can take parameters and can be invoked
- PL/SQL has two types of subprogram called **(stored) procedures** and **functions**
- Procedures and functions are identical except that functions always return a value (procedures do not)
- By processing the SQL code on the database server, both
  - the number of instructions sent across the network and
  - the amount of data returned from the SQL statements are reduced
- A **package** is a collection of procedures, functions, variables, and SQL statements that are grouped together and stored as a single program unit

```
CREATE [OR REPLACE] PROCEDURE <procedureName> (<paramList>) [IS| AS]
<localDeclarations>
BEGIN
<procedureBody>;
END;
/
```



A parameter in the paramList is specified as:

**<name> <mode> <type>**

**Mode:**

**IN** -- input parameter (default)

**OUT** -- output parameter

**INOUT** -- input or output parameter

```
CREATE PROCEDURE remove_emp (employee_id IN Number) AS

BEGIN
    DELETE FROM employees
    WHERE employees.employee_id = remove_emp.employee_id;

END;
/
```

In PL/SQL

‘;’ ends a line without execution

‘/’ execute the command and create the procedure

```
SQL > exec remove_emp (10);
```



## Stored procedures – advantages and disadvantages

- Extensibility
- Reusability
- Maintainability
- Aid abstraction
- Improves testability
  - Can be tested independently of the application
- Speed / optimization
  - Stored procedures are cached on the server
- Improved security

- Limited Coding Functionality
  - Not as robust as app code
- Portability issues
- Reduced flexibility and agility

3.1. Introduction to distributed systems and middleware

3.2. Database-centric architectures

## **3.2.1. Data warehousing and business intelligence**

3.2.2. Big data architectures

3.3. Message-oriented architectures

3.4. Object-oriented architectures

3.5. Component-based architectures

3.6. Service-oriented architectures

3.7. Blockchain-based systems

# Data warehousing and business intelligence

- Business intelligence
  - Extract knowledge from large amounts of collected business data
  - Combination of technologies
    - Data warehousing
    - On-Line Transaction Processing (OLTP)
    - Data mining
    - .....

**Data warehousing** is a collection of methods, techniques and tools which is used to support knowledge workers such as senior managers, directors, managers, and business analysts to conduct data analyses that help with performing decision-making processes and improving information resources.

# Data analysis problems

- The same data found in many different systems
  - Example: customer data across different stores and departments
  - The same concept is defined differently
- Heterogeneous sources
  - Relational DBMS, On-Line Transaction Processing (OLTP)
  - Unstructured data in files (e.g., MS Word)
  - Legacy systems
  - .....
- Data is suited for operational systems
  - Accounting, billing, etc.
  - Do not support analysis across business functions
- Data quality is bad
  - Missing data, imprecise data, different use of systems
- Data is “volatile”
  - Data deleted in operational systems
  - Data change over time – no historical information

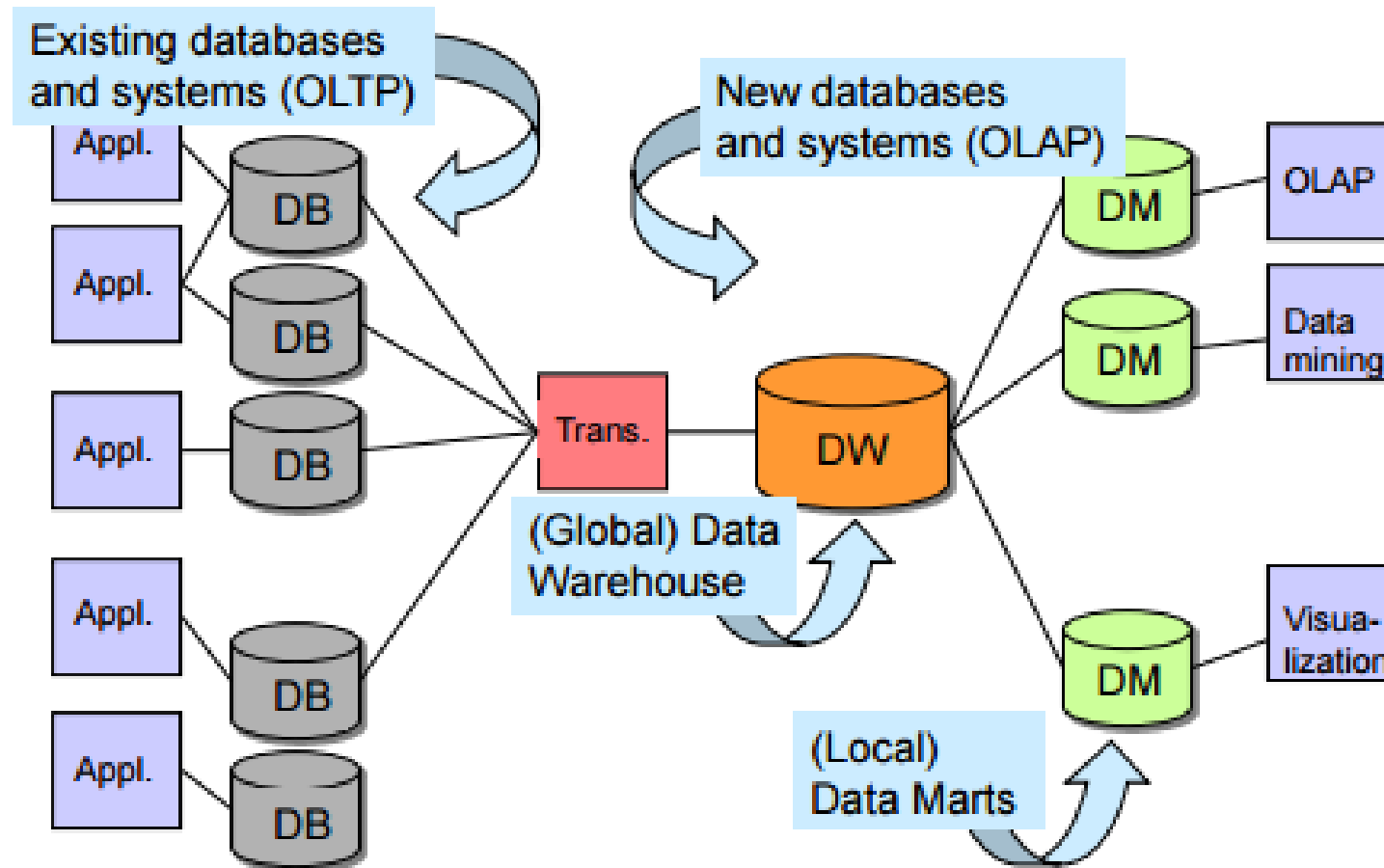
A **data warehouse** is a collection of data that supports decision-making processes.

It provides the following features:

- It is subject-oriented
- It is integrated and consistent
- It shows its evolution over time and it is non-volatile

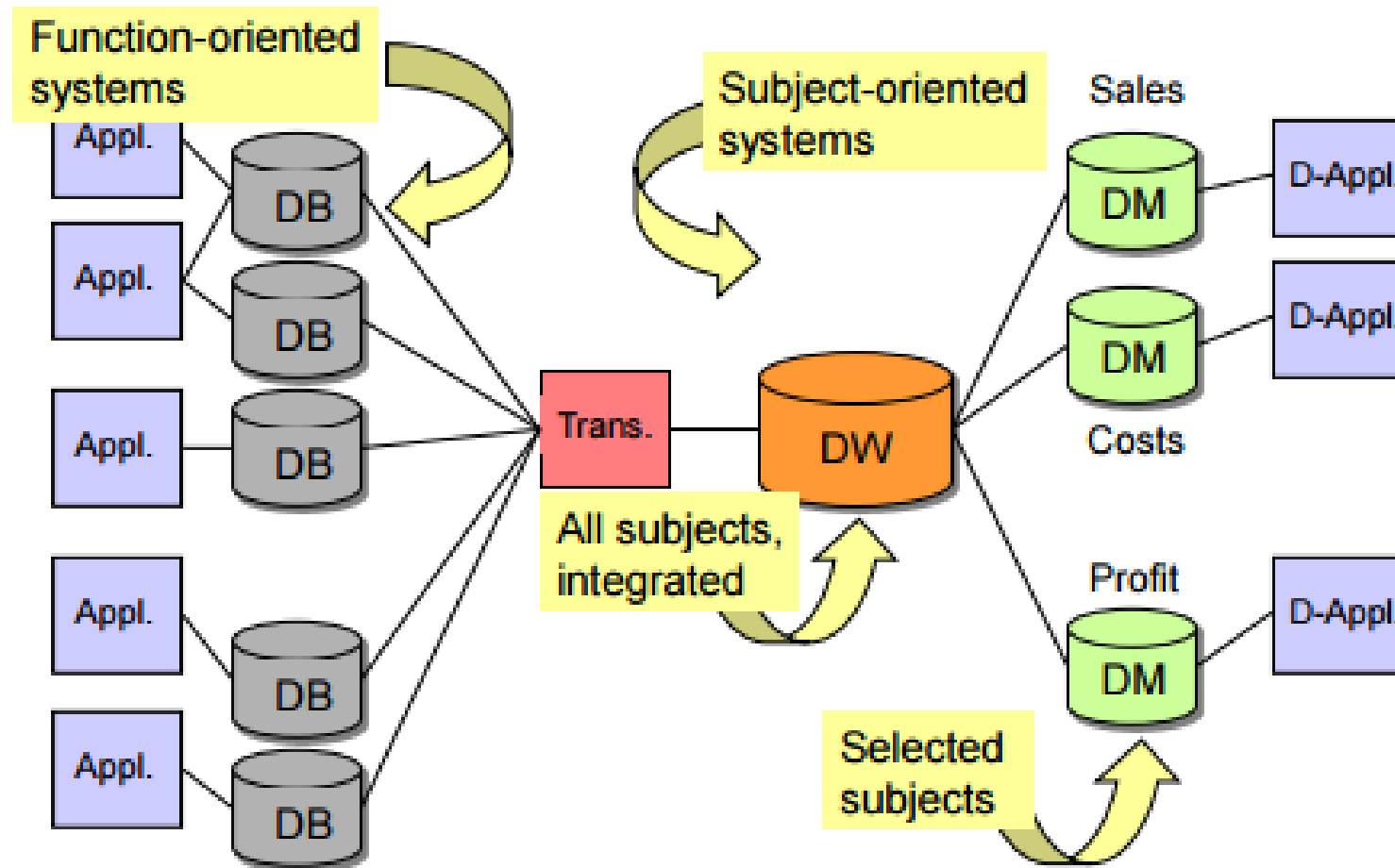
- Data from the operational systems are
  - Extracted
  - Cleansed
  - Transformed
  - Aggregated (?)
  - Loaded into the data warehouse
- A good data warehouse is a prerequisite for successful BI

# Data as materialized views



Analogy: (data) producers ↔ warehouse ↔ (data) consumers

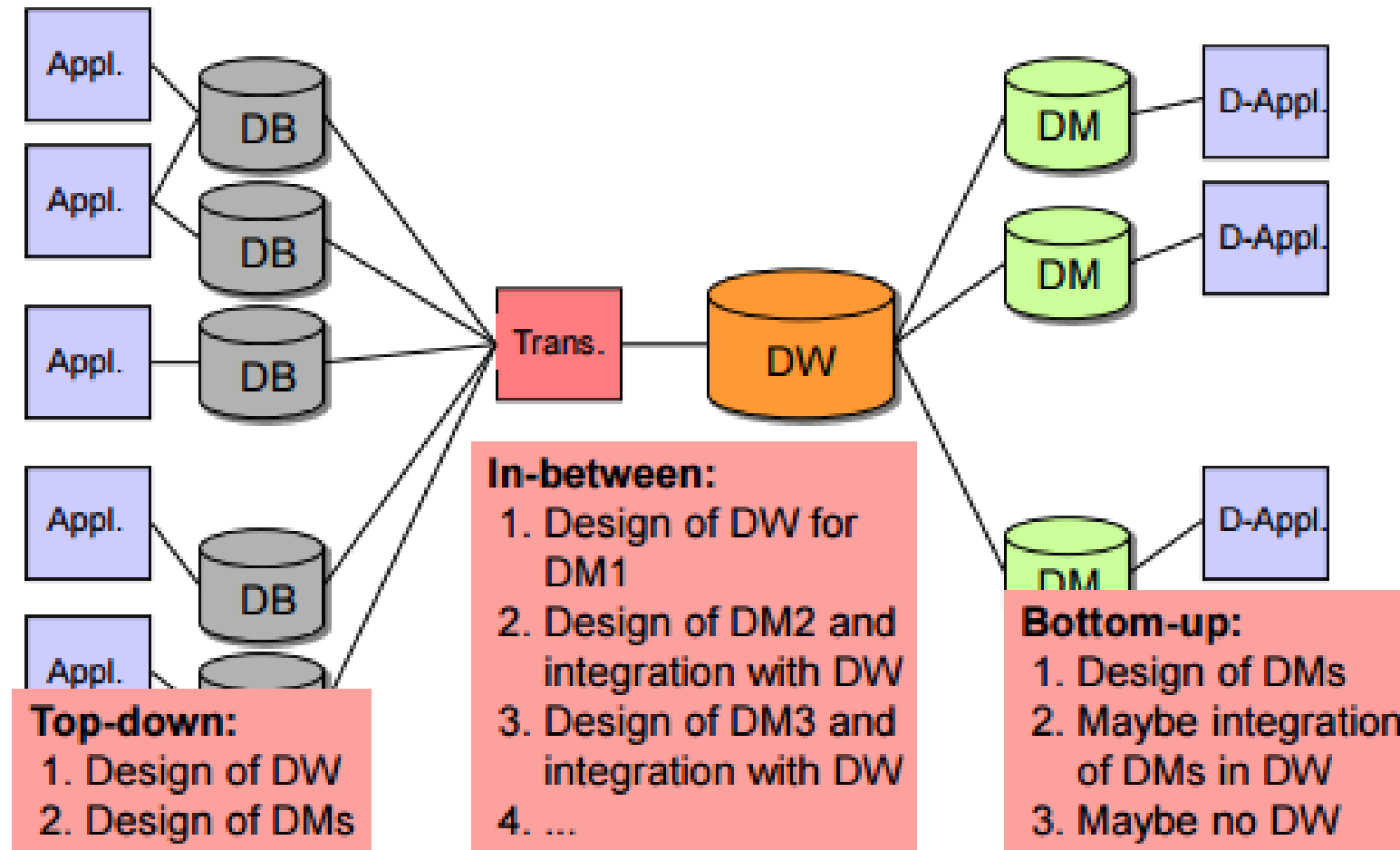
# Function vs. subject orientation



A **data mart** (DM) is a subset or an aggregation of the data stored to a primary data warehouse. It includes information relevant to a specific business area, corporate department, or category of users.

[“Data Warehouse Design: Modern Principles and Methodologies.” Golfarelli, M., & Rizzi, S. (2009)]

# Top-down vs. bottom-up





# Operational database vs Data Warehouse

## OLTP vs OLAP

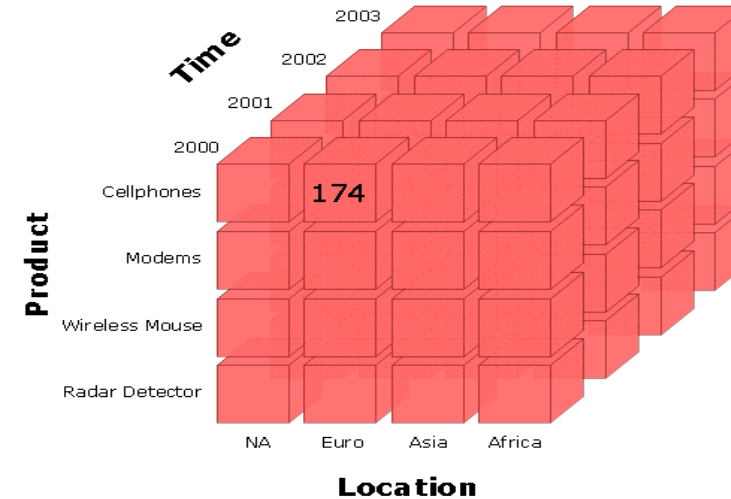
Feature	OLTP	OLAP
Users	Clerk, IT professional	Knowledge worker
Functions	Day to day operations	<b>Decision support</b>
DB Design	Application-oriented	<b>Subject-oriented</b>
Data	Current, isolated	Historical, <b>consolidated</b>
View	Detailed, flat relational	Summarized, <b>multidimensional</b>
Usage	Structured, normalized, repetitive	Ad hoc
Unit of work	Short, simple transaction	<b>Complex query</b>
Access	Read/write	Read mostly
Operations	Index/hash on prim. Key	Lots of scans
# Rec. accessed	Tens	Millions
#Users	Thousands	Hundreds
Db size	100 MB-GB	100GB-TB
Metric	Trans. throughput	Query throughput, response

OLAP - Online analytical processing  
OLTP - Online transaction processing

[“Data Warehouse Design: Modern Principles and Methodologies.” Golfarelli, M., & Rizzi, S. (2009)]

# OLAP data cube

- Data cube
  - Data analysis tool in OLAP
  - Generalized GROUP BY queries
  - Aggregate facts based on chosen dimensions
  - Product, store, time dimensions
  - Sales measure of sale facts
- Why data cube?
  - Good for visualization
  - Multidimensional, intuitive
  - Support interactive OLAP operations
    - Slice
    - Dice
    - Drill down/up
    - Aggregation



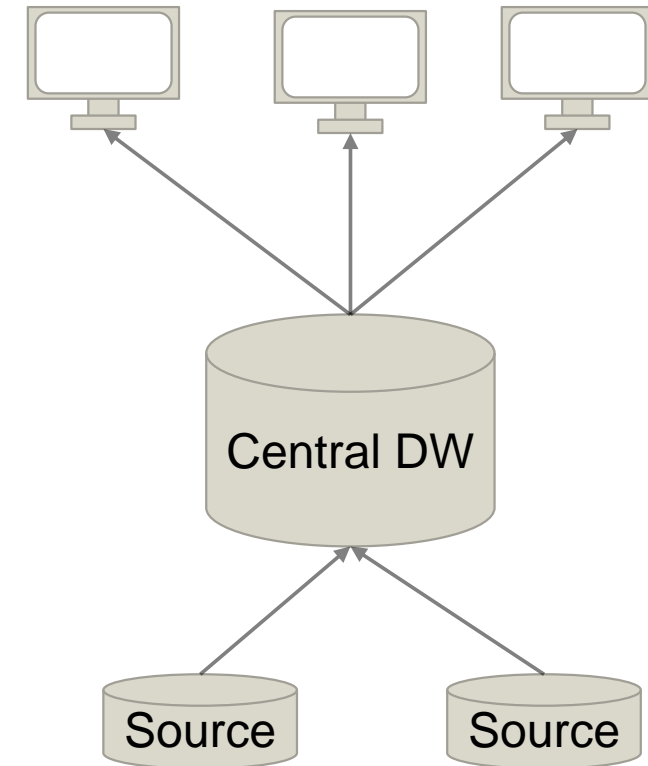
[“Data Warehouse Design: Modern Principles and Methodologies.” Golfarelli, M., & Rizzi, S. (2009)]

# Extract, Transform, Load (ETL)

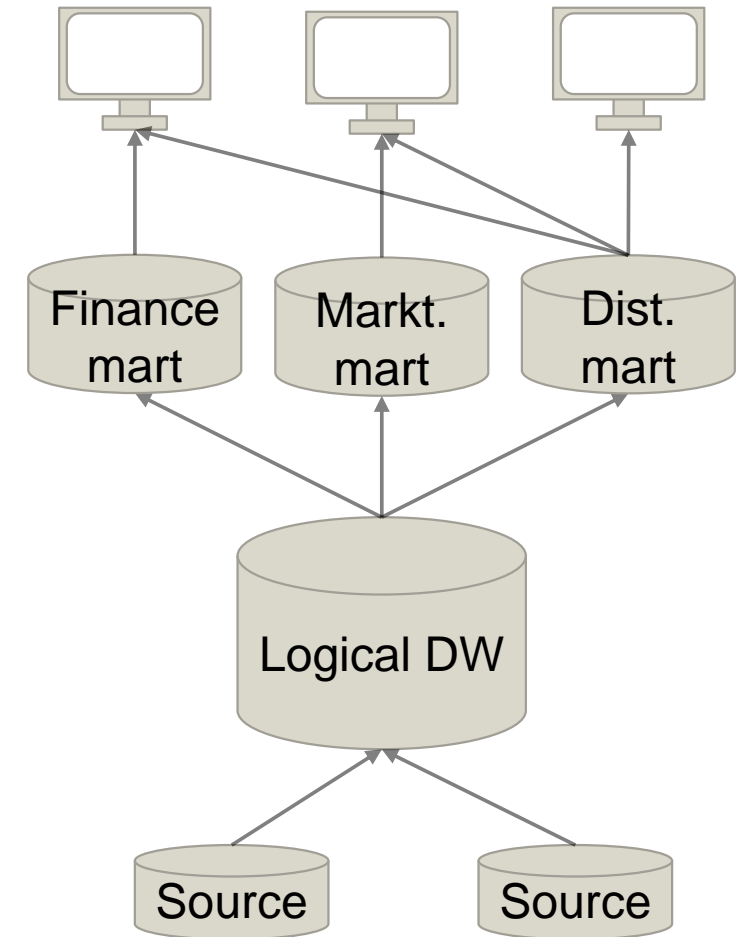
## Getting multidimensional data into the DW

- Problems
  1. Data from different sources
  2. Data with different formats
  3. Handling of missing data and erroneous data
  4. Query performance of DW
- ETL
  - Extract (for problem #1)
  - Transformations / cleansing (for problems #2, #3)
  - Load the data to DW
    - Materialized view - precompute some partial result in advance and store it (problem #4)
- The most time-consuming process in DW development
  - 80% of development time spent on ETL

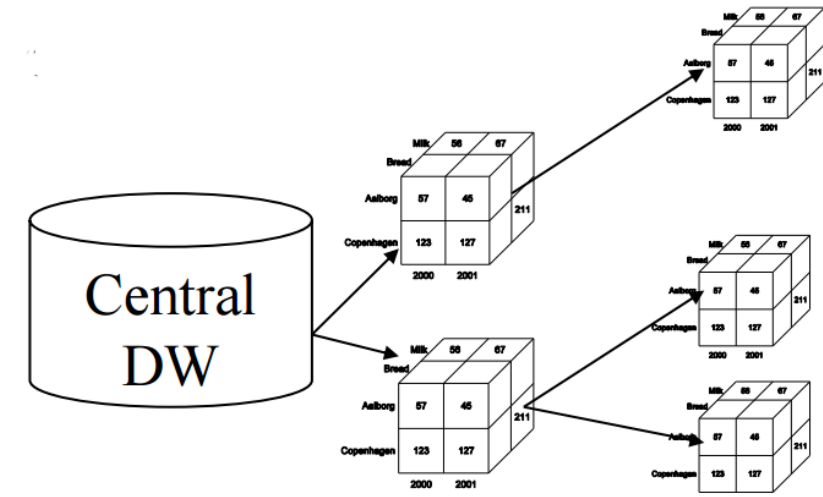
- All data in one, central DW
- All client queries directly on the central DW
- Pros
  - Simplicity
  - Easy to manage
- Cons
  - Bad performance due to lack of redundancy/workload distribution



- Data stored in separate data marts, aimed at special departments
- Logical DW (i.e., virtual)
- Data marts contain detail data
- Pros
  - Performance due to distribution
- Cons
  - More complex



- Central DW is materialized
- Data is distributed to data marts in one or more tiers
- Only aggregated data in cube tiers
- Data is aggregated/reduced as it moves through tiers
- Pros
  - Best performance due to redundancy and distribution
- Cons
  - Most complex
  - Hard to manage



- Metadata management
  - Need to understand data = metadata needed
  - Greater need in OLAP than in OLTP as “raw” data is used
  - Need to know about:
    - Data definitions, dataflow, transformations, versions, usage, security
- DW project management
  - DW projects are large and different from ordinary SW projects
    - 12-36 months and US\$ 1+ million per project
    - Data marts are smaller and “safer” (bottom up approach)
- Reasons for failure
  - Lack of proper design methodologies
  - High HW+SW cost
  - Deployment problems (lack of training)
  - Organizational change is hard... (new processes, data ownership,..)
  - Ethical issues (security, privacy,...)

# Other challenges

Security of data  
during ETL

Data visualizations

Decision analysis  
(what-if analysis)

...



3.1. Introduction to distributed systems and middleware

3.2. Database-centric architectures

3.2.1. Data warehousing and business intelligence

**3.2.2. Big data architectures**

3.2.2.1. Scalability

3.2.2.2. NoSQL data stores

3.2.2.3. Data processing

3.3. Message-oriented architectures

3.4. Object-oriented architectures

3.5. Component-based architectures

3.6. Service-oriented architectures

3.7. Blockchain-based systems

Big data refers to datasets whose size is beyond the ability of typical database software tools to capture, store, manage, and analyze.

[*"Big data: The next frontier for innovation, competition, and productivity."* Manyika, James, et al. (2011)]

Big data is a loosely defined term used to describe data sets so large and complex that they become awkward to work with using standard statistical software.

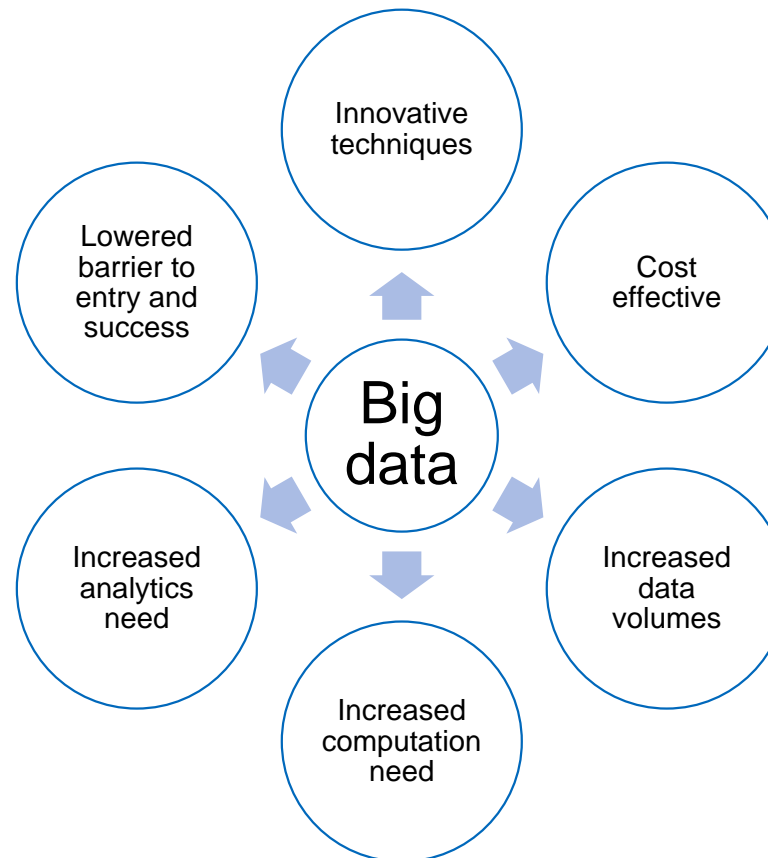
[*"Big Data: Big gaps of knowledge in the field of Internet"*. Snijders, C. et al. (2012)]

Big data is about "What", not "why". We don't always need to know the cause of a phenomenon; rather, we can let data speak for itself.

[*"Big data, a revolution that will transform how we live, work and think."* Mayer-Schönberger and Cukier (2013)]

“Big data is high-**volume**, high-**velocity** and high-**variety** information assets that demand **cost-effective**, **innovative** forms of information processing for **enhanced insight** and **decision making**.”

[Gartner]



[“Big data analytics.” David Loshin (2013)]

### Every minute:

2014

Facebook users share **nearly 2.5 million** pieces of content.

Twitter users tweet **nearly 300,000 times**.

Instagram users post **nearly 220,000 new photos**.

YouTube users upload **72 hours of new video content**.

Apple users download **nearly 50,000 apps**.

Email users send over **200 million messages**.

Amazon generates over **\$80,000 in online sales**.

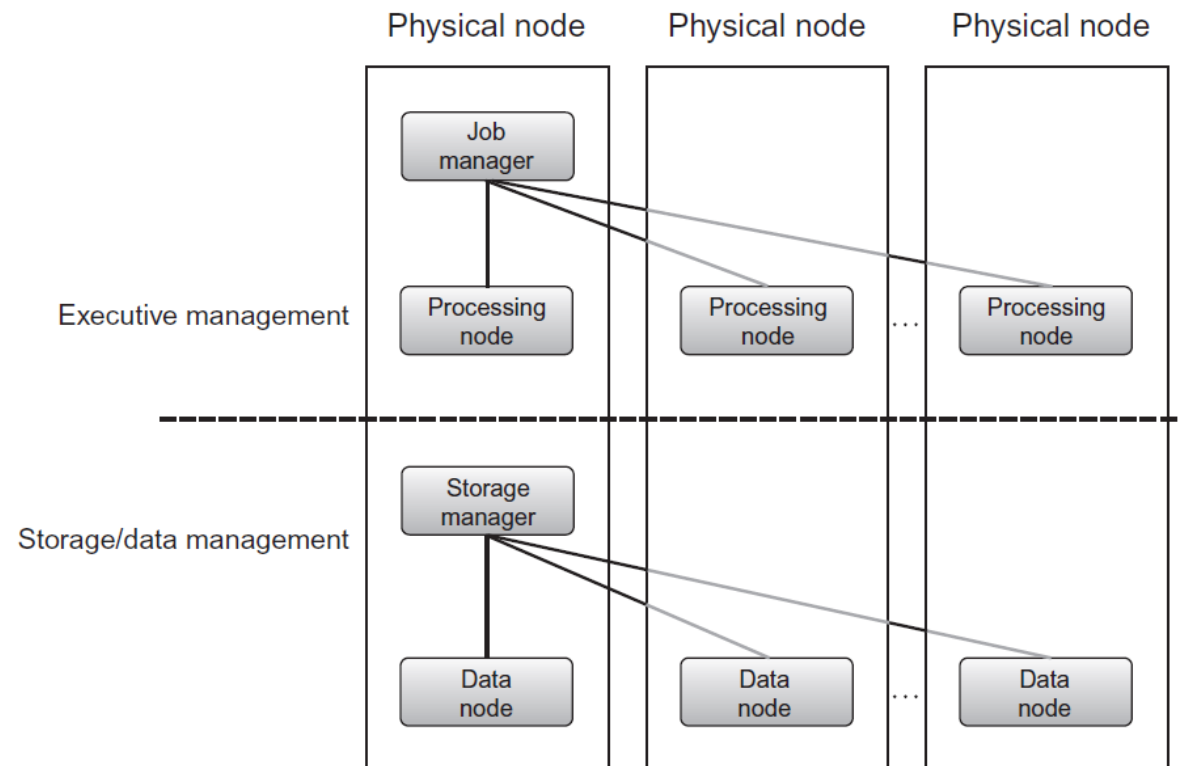
“Using all available data is feasible in an increasing number of contexts. But it comes at a cost.”

[“Big data, a revolution that will transform how we live, work and think.”Mayer-Schönberger and Cukier (2013)]

# Big data

## Key computing resources

- Processing capability: CPU, processor, or node.
- Memory
- Storage
- Network



Typical organization of resources in a big data platform

- One of the key concepts driving the requirement behind big data is: **scalability**
  - Therefore it is highly important to understand the storage mechanisms of what we count as big data
  - Understand the challenges this requirement imposes on the architecture of systems
  - Enable programmers to use these architectures without too much overhead when creating systems that harness the power and value of big data

3.1. Introduction to distributed systems and middleware

3.2. Database-centric architectures

3.2.1. Data warehousing and business intelligence

3.2.2. Big data architectures

## **3.2.2.1. Scalability**

3.2.2.2. NoSQL data stores

3.2.2.3. Data processing

3.3. Message-oriented architectures

3.4. Object-oriented architectures

3.5. Component-based architectures

3.6. Service-oriented architectures

3.7. Blockchain-based systems

***Scalability*** is the ability of a system, network, or process to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth.

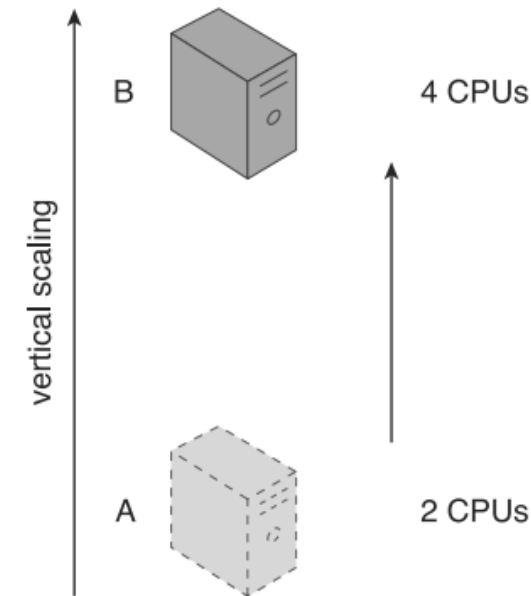
["Characteristics of scalability and their impact on performance." Bondi, André B. (2000)]



# Vertical scaling

When an existing IT resource is replaced by another with higher or lower capacity, vertical scaling is said to have occurred

- Specifically, the replacing of an IT resource with another that has a higher capacity is referred to as **scaling up**
- Replacing an IT resource with another that has a lower capacity is considered **scaling down**
- **Pro:** it is very easy to design for vertical scaling  
→if you're certain about the ceiling for your application's usage, then vertical scaling can be a nice fast alternative to building truly scalable systems.

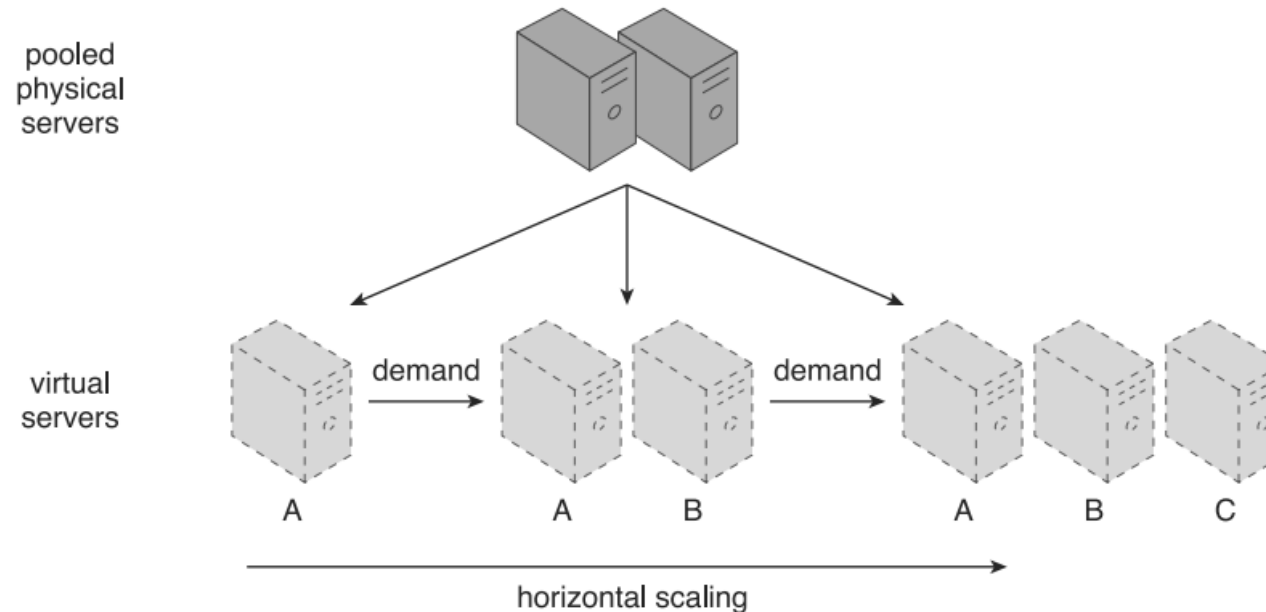


[“Cloud computing - concepts, technology & architecture.” Thomas Erl et al. (2013)]

# Horizontal scaling

The allocating or releasing of IT resources that are of the same type is referred to as horizontal scaling.

- The horizontal **allocation of resources** is referred to as **scaling out**
- Horizontal **releasing of resources** is referred to as **scaling in**
- Horizontal scaling is a common form of scaling within cloud environments

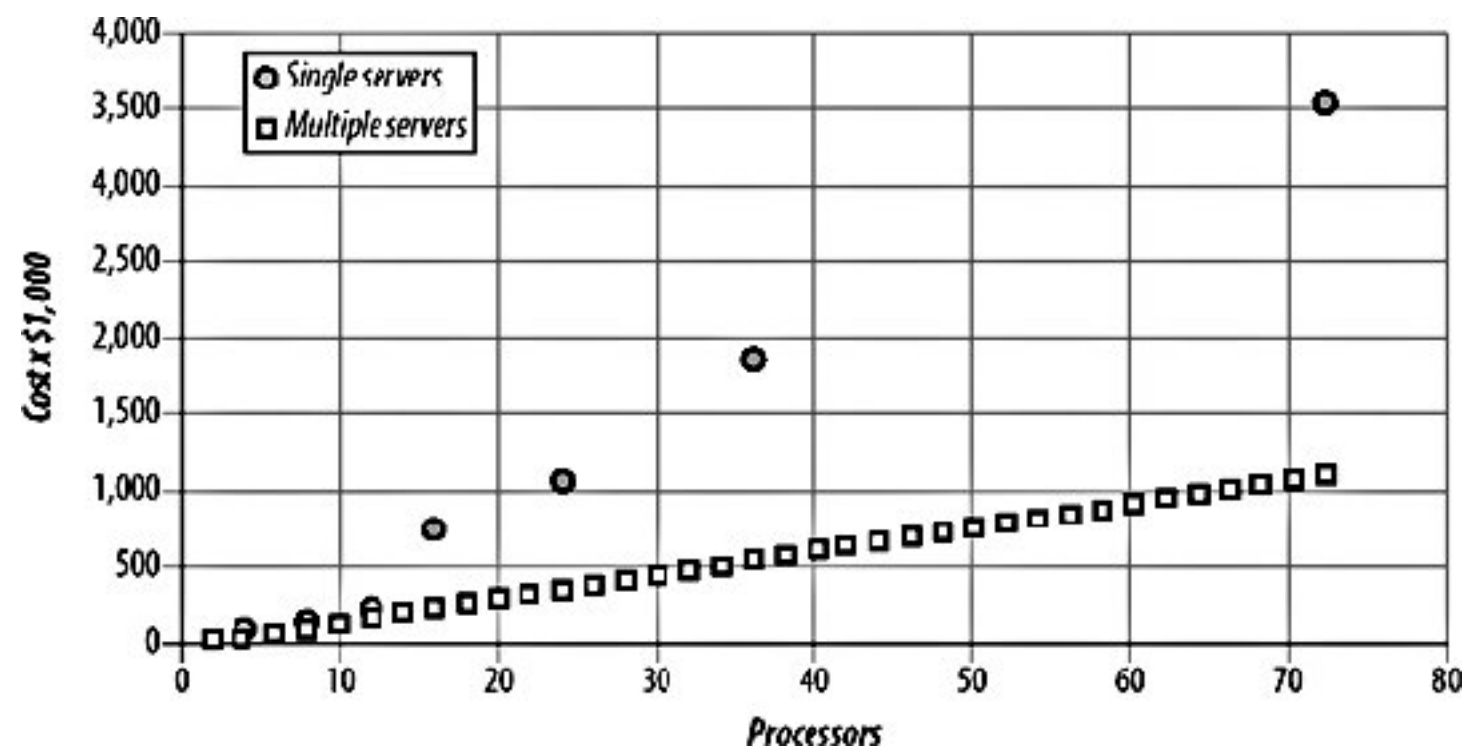


[“Cloud computing - concepts, technology & architecture.” Thomas Erl et al. (2013)]

# Horizontal vs vertical scaling

Horizontal Scaling	Vertical Scaling
less expensive (through commodity hardware components)	more expensive (specialized servers)
IT resources instantly available	IT resources normally instantly available
resource replication and automated scaling	additional setup is normally needed
additional IT resources needed	no additional IT resources needed
not limited by hardware capacity	limited by maximum hardware capacity

# Horizontal and vertical scaling

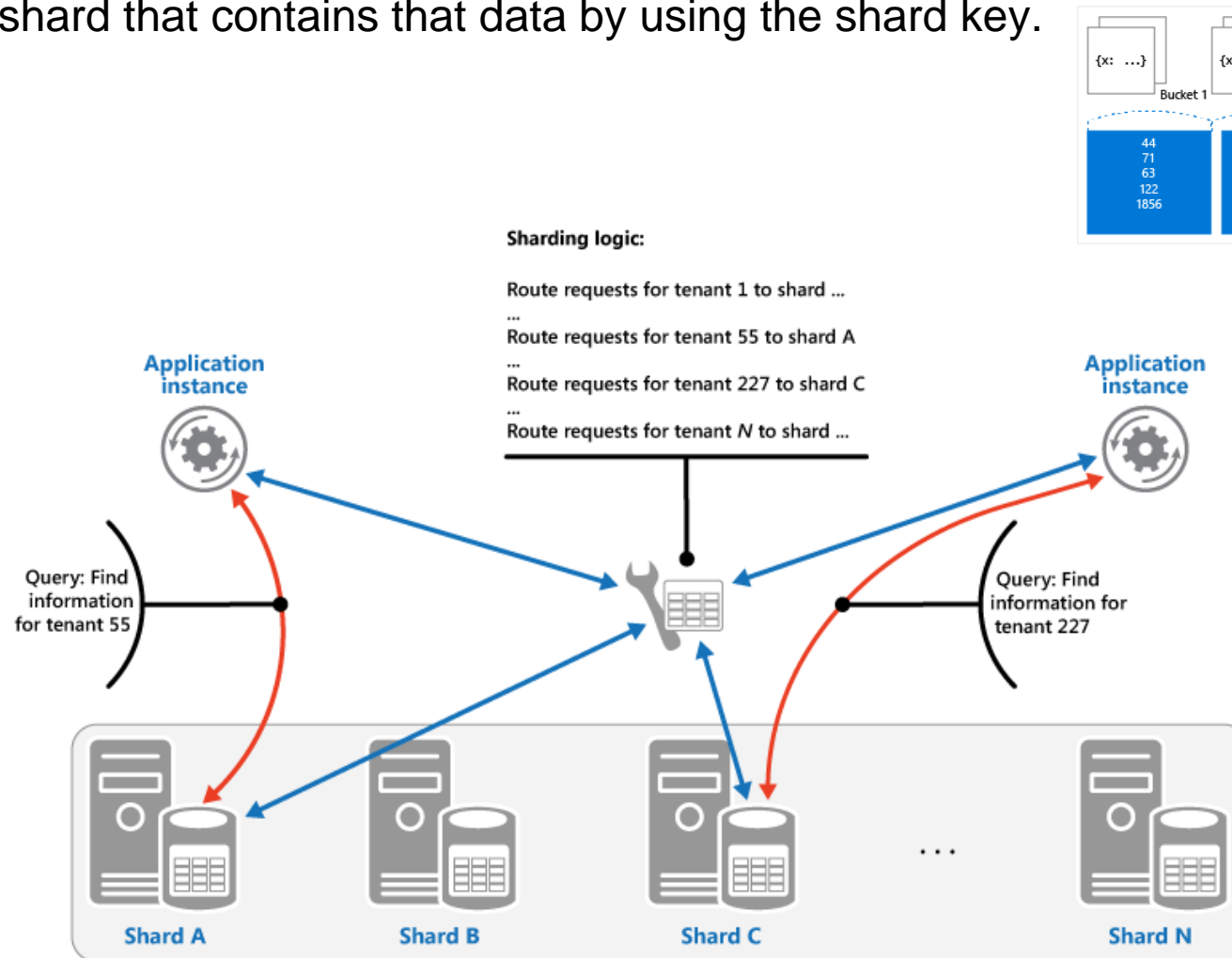


[“Building scalable web sites - building, scaling, and optimizing the next generation of web applications.” Cal Henderson (2006)]

## Sharding Pattern

- Divide the data store into horizontal partitions or shards. Each shard has the same schema, but holds its own distinct subset of the data
  - A shard is a data store in its own right (it can contain the data for many entities of different types), running on a server acting as a storage node
- 
- Same schema, partition by some primary field (like tenant id, user id)
  - Place each shard on a separate node in a cluster
  - The cluster spreads the read and write operations
  - Better cache locality
  - Requires central lookup registry (e.g., hash table) to map requests to shards

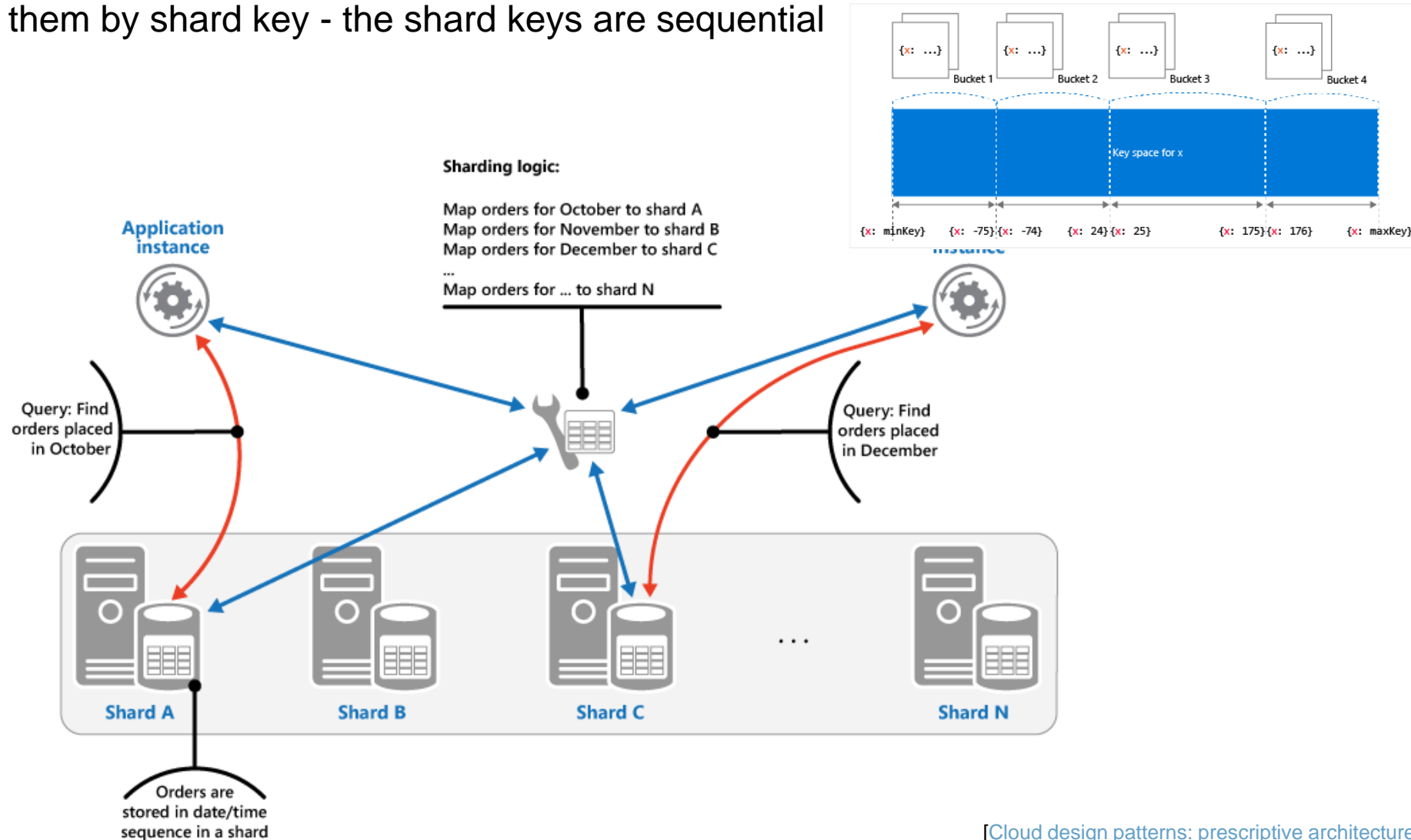
The sharding logic implements a map that routes a request for data to the shard that contains that data by using the shard key.



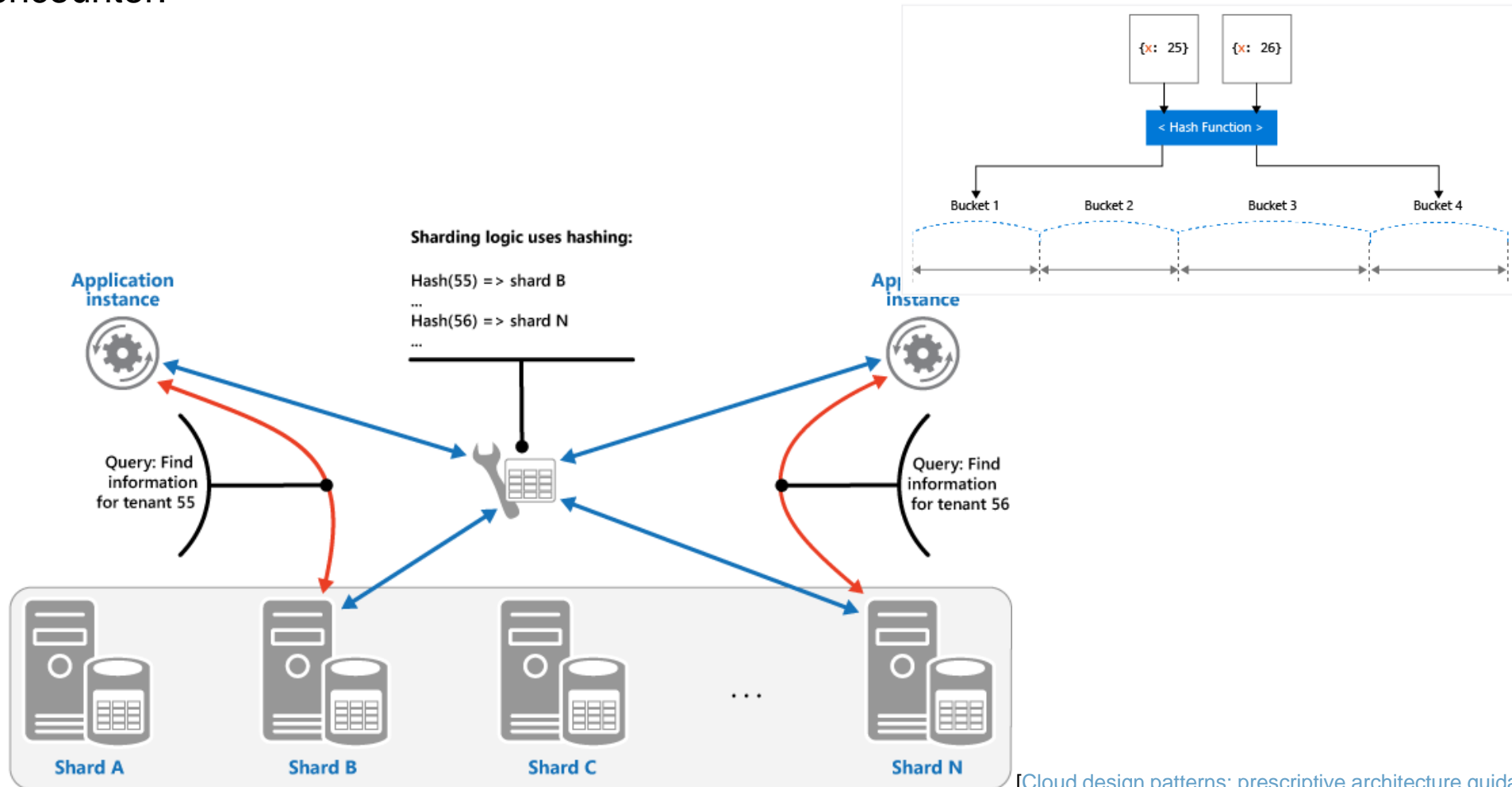
# Sharding

## Range strategy

This strategy groups related items together in the same shard, and orders them by shard key - the shard keys are sequential



The purpose of this strategy is to reduce the chance of hotspots in the data. It aims to distribute the data across the shards in a way that achieves a balance between the size of each shard and the average load that each shard will encounter.





3.1. Introduction to distributed systems and middleware

3.2. Database-centric architectures

3.2.1. Data warehousing and business intelligence

3.2.2. Big data architectures

3.2.2.1. Scalability

**3.2.2.2. NoSQL data stores**

3.2.2.3. Data processing

3.3. Message-oriented architectures

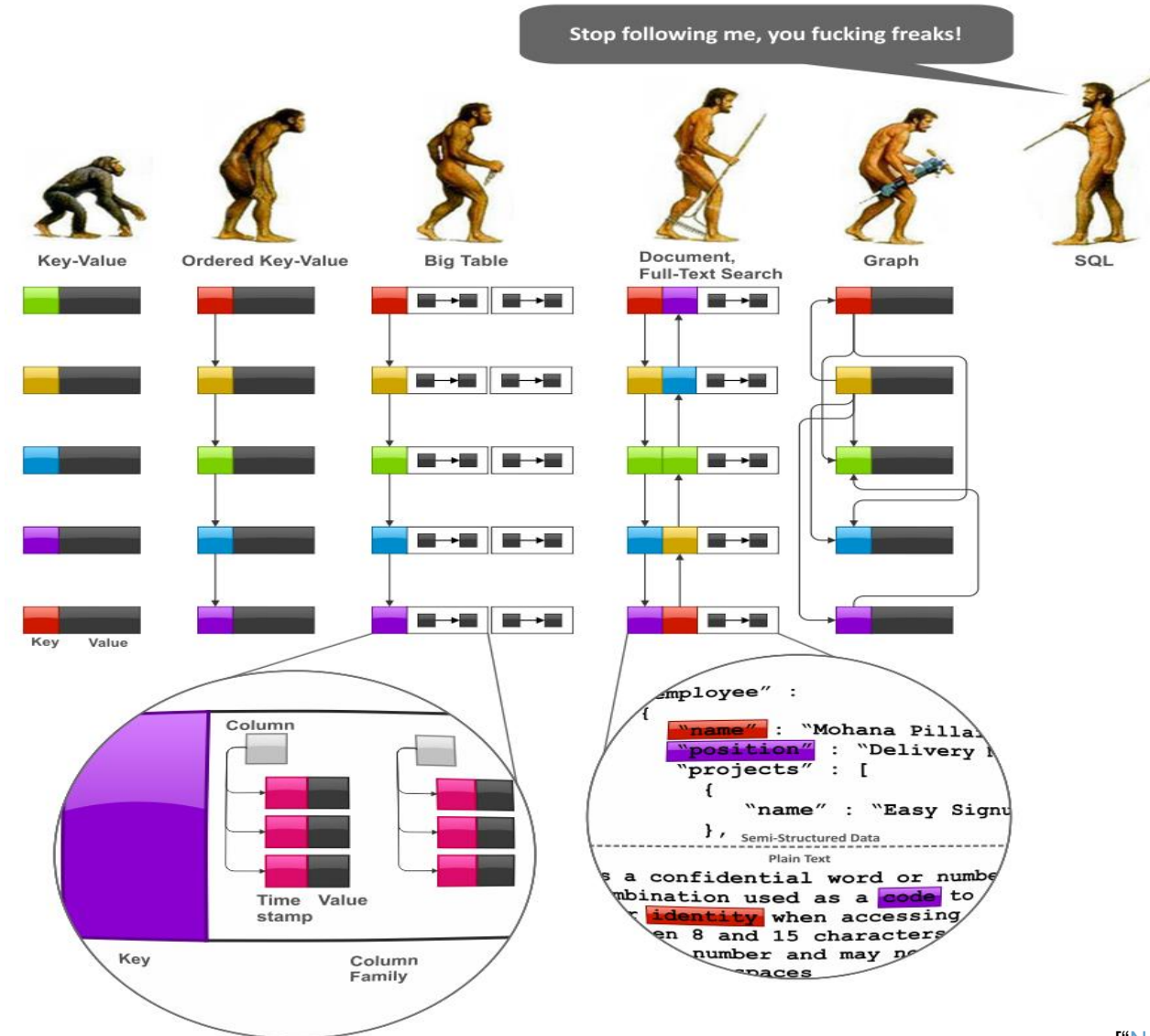
3.4. Object-oriented architectures

3.5. Component-based architectures

3.6. Service-oriented architectures

3.7. Blockchain-based systems

# Storage of big data - overview



[["NoSQL data modeling techniques."](#) Ilya Katsov (2012)]

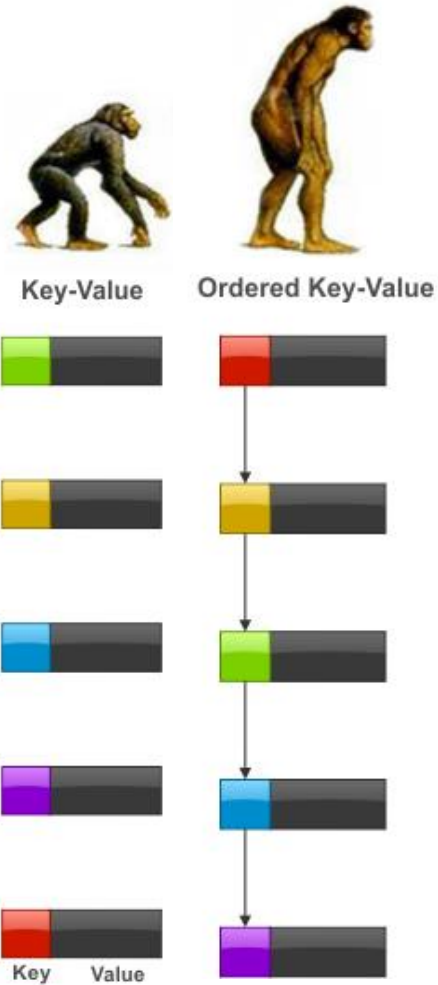
Next generation databases mostly addressing some of the points: being **non-relational, distributed, open-source** and **horizontally scalable**.

[Stefan Edlich (2009)]

The original intention has been **modern web-scale databases**. The movement began early 2009 and is growing rapidly. Often more characteristics apply such as:

- Schema-free
- Easy replication support
- Simple API
- Eventually consistent / BASE (not ACID)
- A huge amount of data

So the misleading term "*nosql*" (the community now translates it mostly with "**not only sql**") should be seen as an alias to something like the definition above.



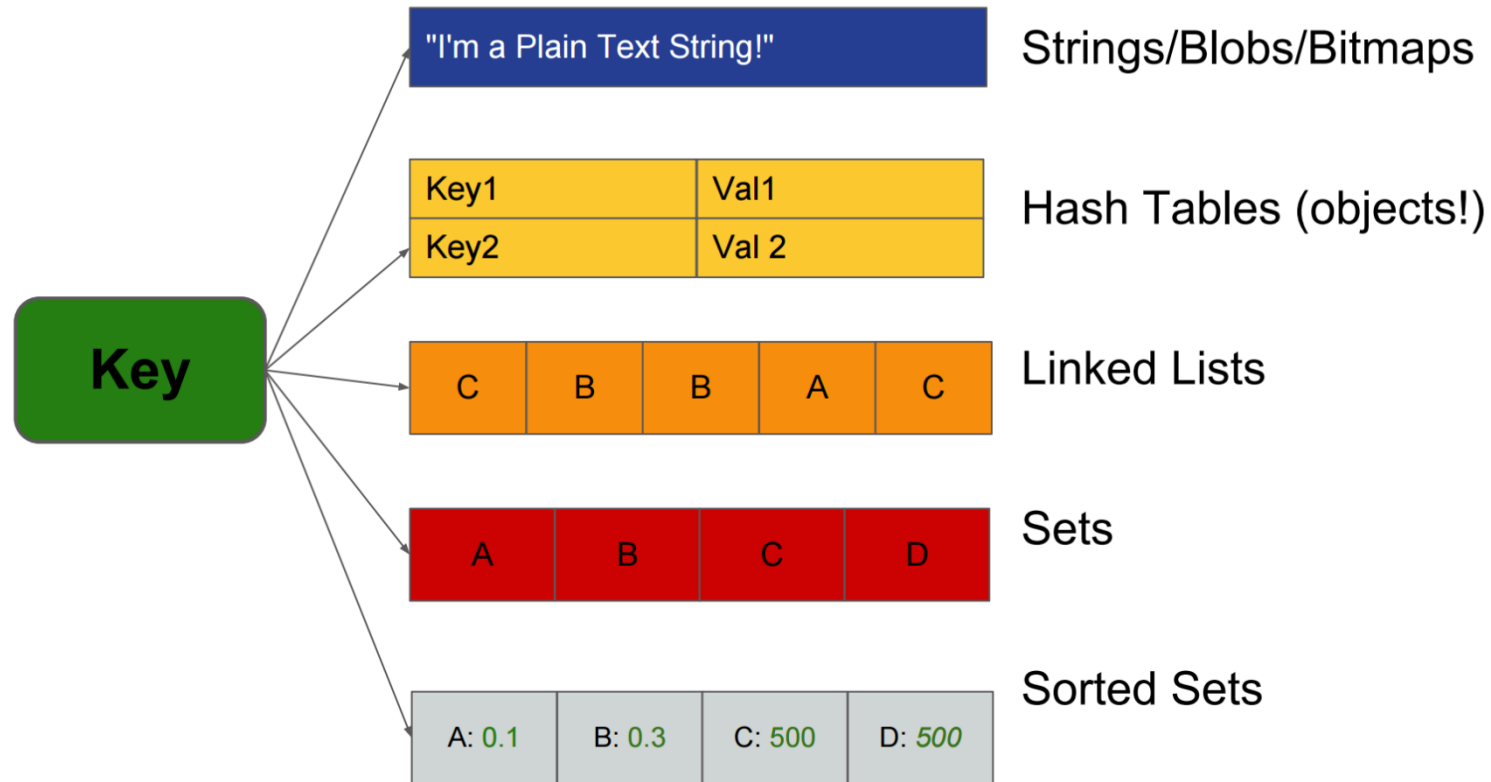
**Key-value store** – data is stored in unstructured records consisting of a **key** + the **values** associated with that record.

[[Aerospike](#)]

- Key-value model has poor performance to cases that require processing of key ranges. Ordered Key-value model overcomes this limitation and significantly improves aggregation capabilities.
- Ordered key-value model is very powerful, but it does not provide any framework for value modeling.
- Some key-value stores like Oracle Coherence gradually move towards document databases via addition of indexes and in-database entry processors.

[[“NoSQL data modeling techniques.”](#) Ilya Katsov (2012)]

“Redis is an *open source*, networked, *blazing fast in-memory data structure store*, used as database, cache and message broker.”



For list of operations visit <http://redis.io/commands>

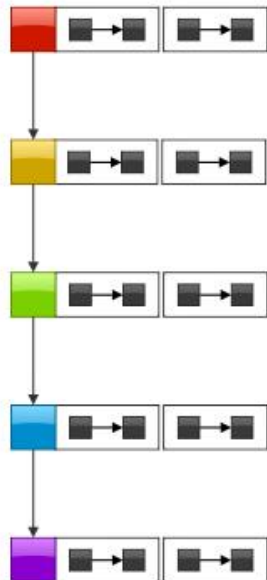
- Example applications
  - Storing real-time stock prices
  - Cache management, session management, ...
  - Circular log buffers (call center logs)
  - Real-time analytics
  - Leaderboards (sorted sets for maintaining high-score tables)
  - Real-time communication and broadcasting updates
  - Popular [www.hurl.it](http://www.hurl.it) built on redis
  - Many more ..

- Disk-backed, in-memory database
- Master-slave replication, automatic failover
- Support for multiple data types
- Lua scripting capabilities
- Transaction support
- Values can be set to expire
- Pub/sub to implement messaging

- Dataset size limited to computer RAM (but can span multiple machines' RAM with clustering)
- Not suitable for complex applications with complex data models
- Not suited for interconnected (graph) data
- As the volume of data increases maintaining *unique keys* becomes more difficult and requires some complexity in generating character strings that will remain unique over a large set of keys



Big Table



Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance.

Bigtable, maps two arbitrary

- string values (row key and column key) and
- timestamp (hence three-dimensional mapping)

into an **associated arbitrary byte array**.

It is not a relational database and can be better defined as a sparse, distributed multi-dimensional sorted map.

`(row: string, column: string, time: int64) --> string`

[[Google cluster computing faculty training workshop](#)]

[“Bigtable: a distributed storage system for structured data.” Fay Chang et al.]

### Row

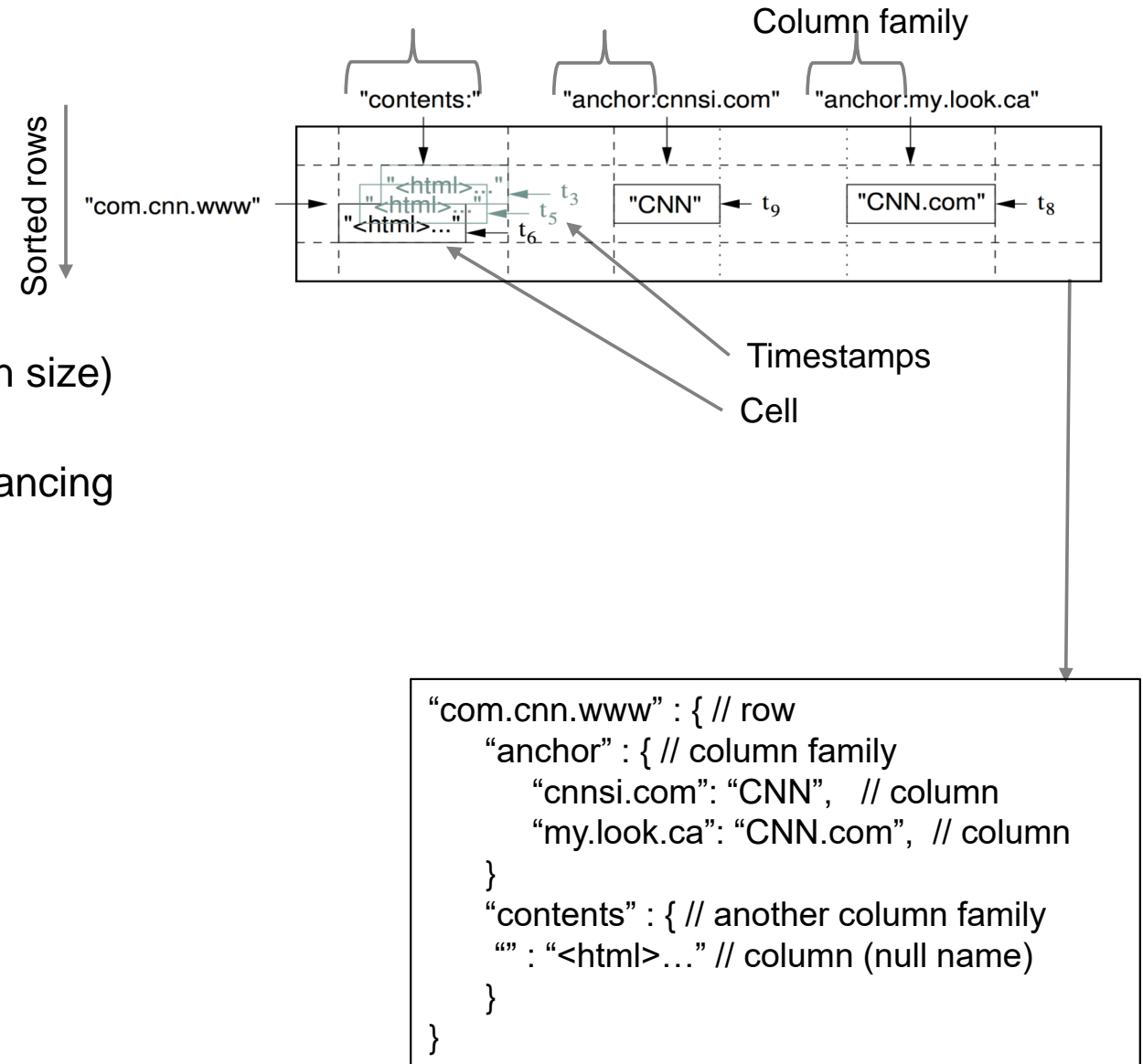
- Data is sorted lexicographically by row key (64kb in size)
- Row key range broken into tablets
  - A tablet is the **unit of distribution** and load balancing

### Column

- Column names of the form *family:qualifier*;  
possibly different #columns per row  
(but column families need to be known)

### Timestamps

- Each cell can be versioned
- Each new version increments the timestamp
- Policies
  - “keep only latest n versions”
  - “keep only versions since time t”





## Basic operations using write API

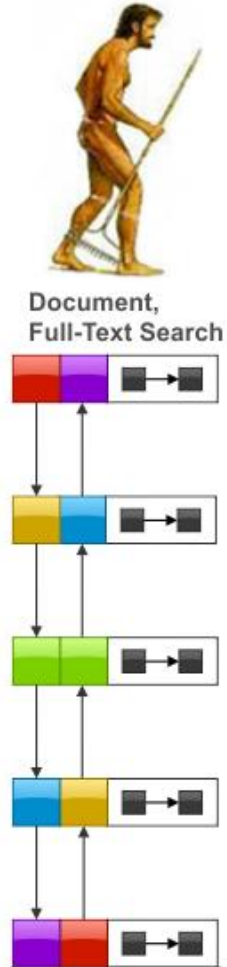
```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);  //atomic row modification
```

- **Performance** of queries in petabytes of data
- **Scalability** by adding horizontally systems
- **Availability** across the globe
- **Flexibility** because no standardized schema
- **High speed retrieval** independent of location
- **Data reliability** even when disks fail
- **Storage size** by harnessing the power of cheap disks
- **Versioning** of data
- **Suitable** for many additions but few modifications of data

- No support for (RDBMS-style) multi-row transactions
- Offers no consistency guarantees for multi-row updates or cross-table updates (not even eventually)
- Lacks the freeform nature of json documents
- No join functionality
- **Suitable** for many additions but few modifications of data

["Bigtable: a distributed storage system for structured data." Fay Chang et al.]

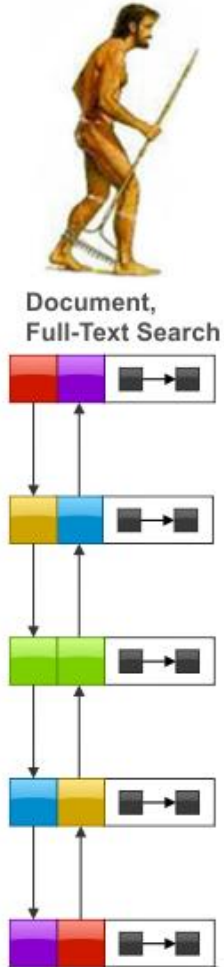


Agile development methods mean that the database schema needs to change rapidly as demands evolve.

SQL databases require their structure to be specified in advance, which means any changes to the information schema require time-consuming ALTER statements to be run on a table.

One of the most popular ways of storing data is a document data model, where each record and its associated data is thought of as a “document.”

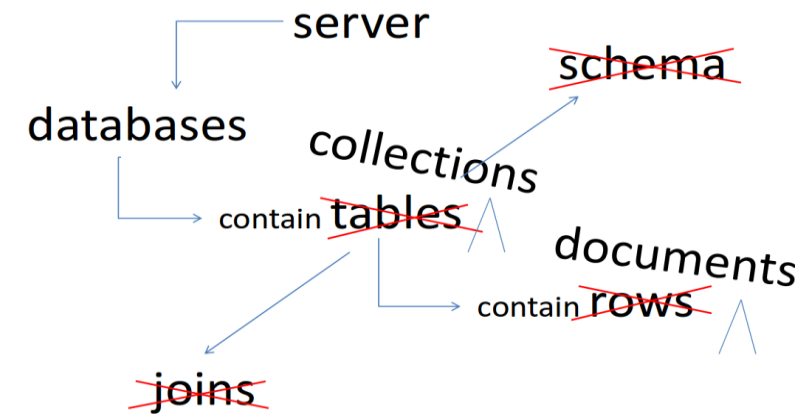
In a document-oriented database, such as MongoDB, everything related to a database object is encapsulated together.



Storing data in this way has the following advantages:

- Documents are independent units which makes performance better (related data is read contiguously off disk) and makes it easier to distribute data across multiple servers while preserving its locality.
- Application logic is easier to write. You don't have to translate between objects in your application and SQL queries, you can just turn the object model directly into a document.
- Unstructured data can be stored easily, since a document contains whatever keys and values the application logic requires. In addition, costly migrations are avoided since the database does not need to know its information schema in advance.

RDBMS	MongoDB
Database	Database
Table	Collection
Row	Document
Column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key: _id)
Partition	Shard
Partition Key	Shard Key
<b>Database server and client</b>	
mysql	mongod
mysql	mongo



- Collections do not enforce a schema.
- Collections can be created on demand.
- A document is a set of key-value pairs and can have dynamic schema.
- Typically, all documents within a collection are of similar or related purpose.

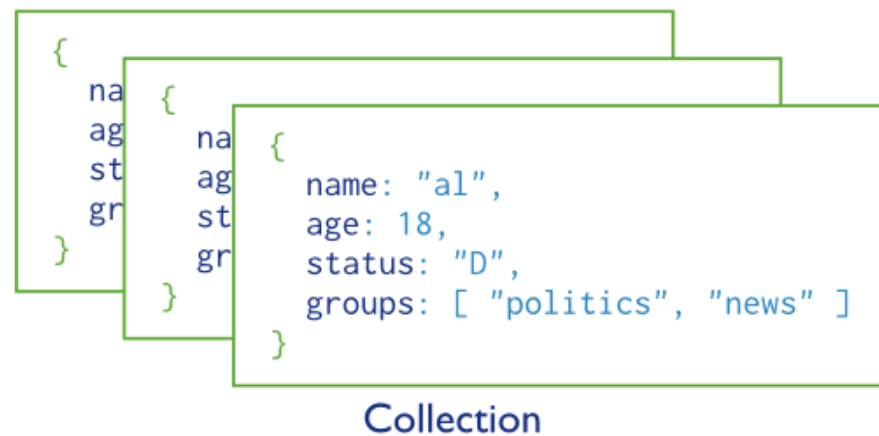
# Document-oriented databases

## Sample document and collection



```
{
  name: "sue",
  age: 26,
  status: "A",
  groups: [ "news", "sports" ]
}
```

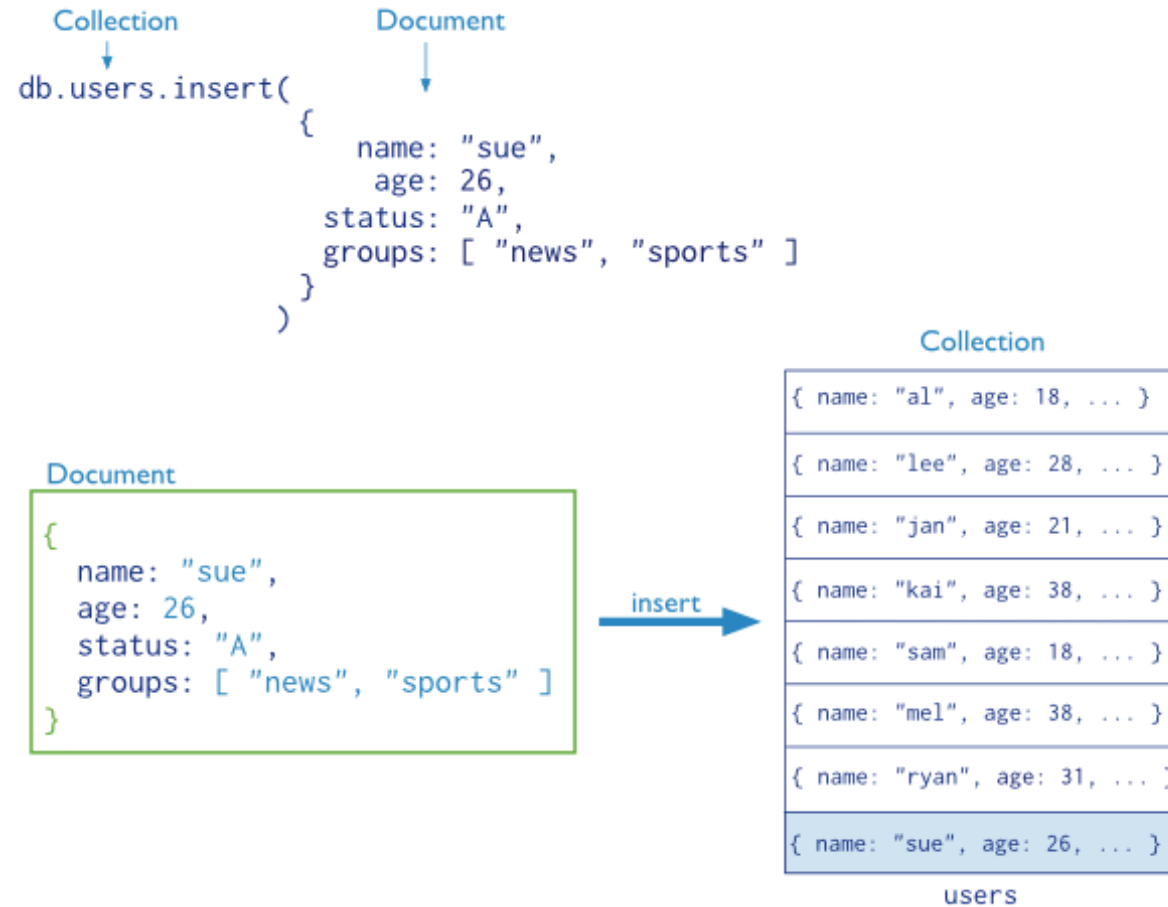
← field: value  
← field: value  
← field: value  
← field: value



String	Integer
Boolean	Double
Arrays	Timestamp
Object	Null
Symbol	Date
Object ID	Binary Data
JS Code	Reg Expressions

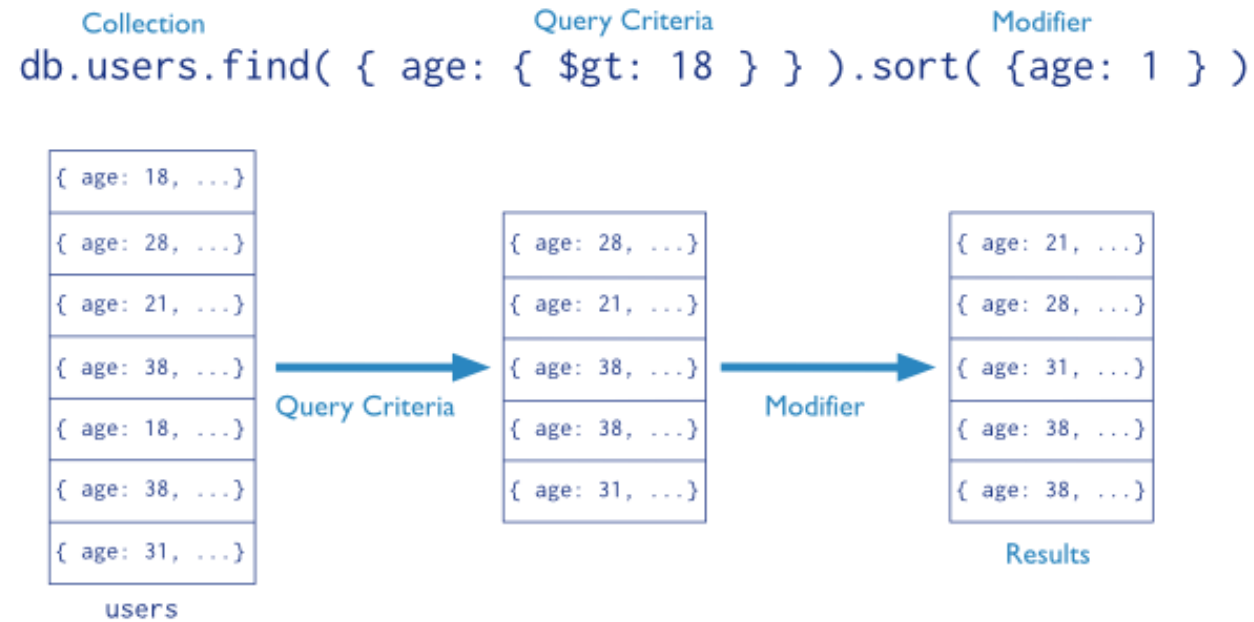
# Document-oriented databases

## MongoDB – inserting document into collection



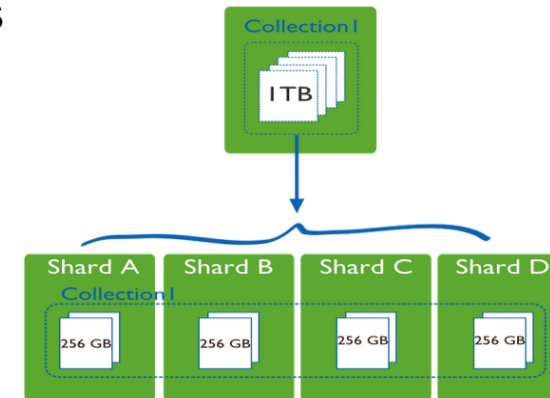
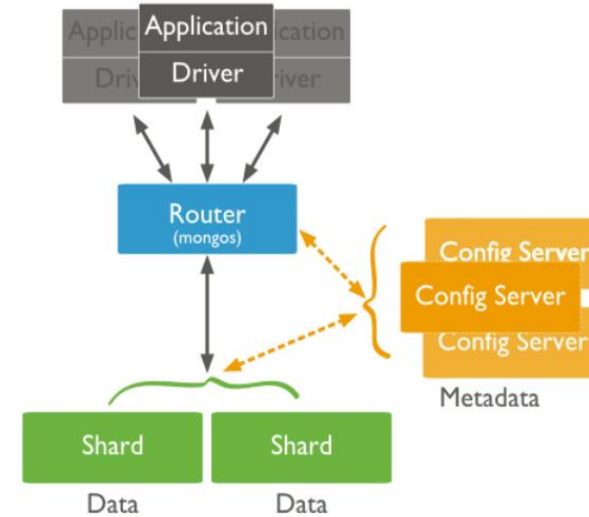
# Document-oriented databases

## MongoDB – querying for data





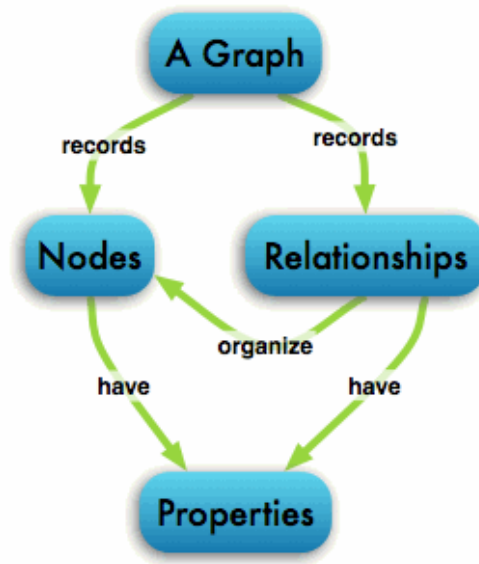
- Sharded cluster comprises of:
  - Shards
  - Query routers (Mongos)
  - Config server
- Shard keys
  - Range based sharding
  - Hash based sharding
- Automatic splitting and balancing of documents across shards





Graph

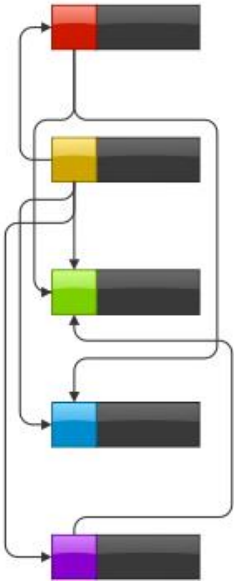
While other databases compute relationships expensively at query time, a graph database stores connections as first class citizens, readily available for any “join-like” navigation operation. Accessing those already persistent connections is an efficient, constant-time operation and allows you to quickly traverse millions of connections per second per core.



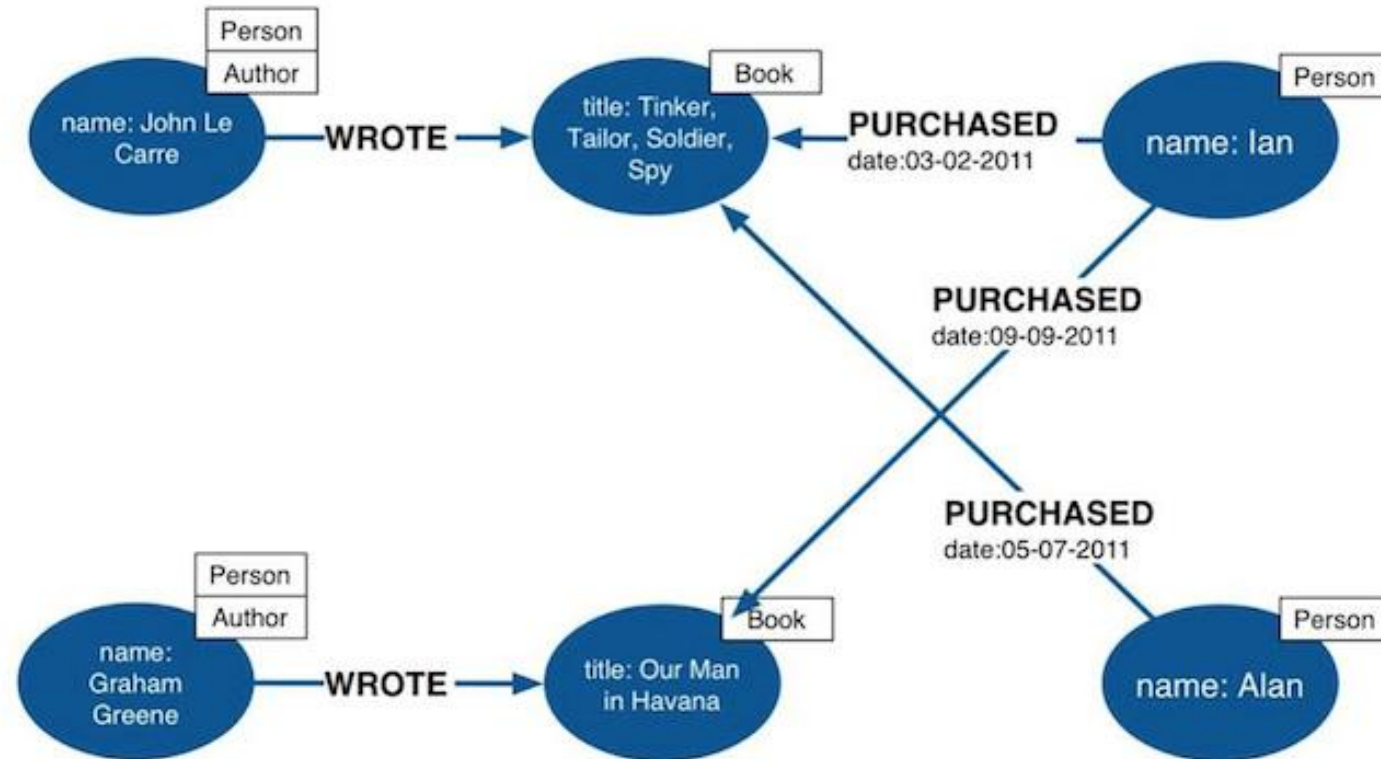
- There is one core consistency rule in a graph database: “No broken links”.
- Since a relationship always has a start and end node, you can’t delete a node without also deleting its associated relationships.



Graph



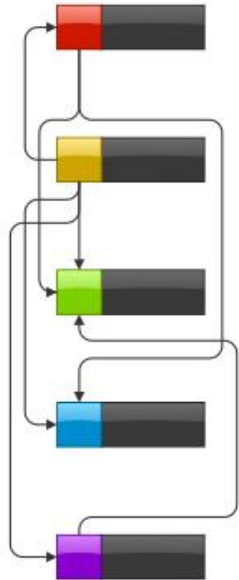
# Labeled Property Graph Data Model



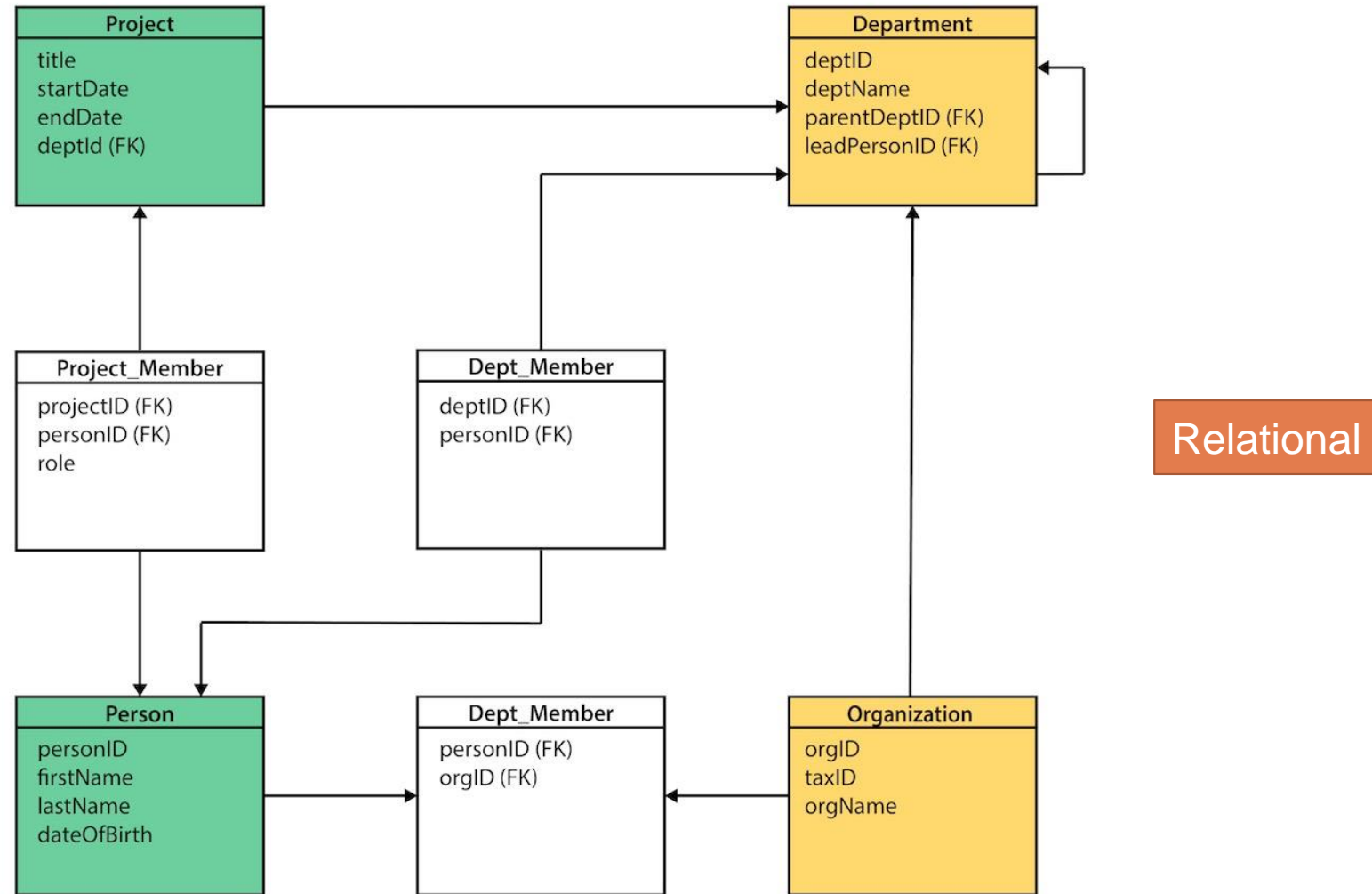
[What is a graph database - [Neo4j developer documentation](#)]



Graph

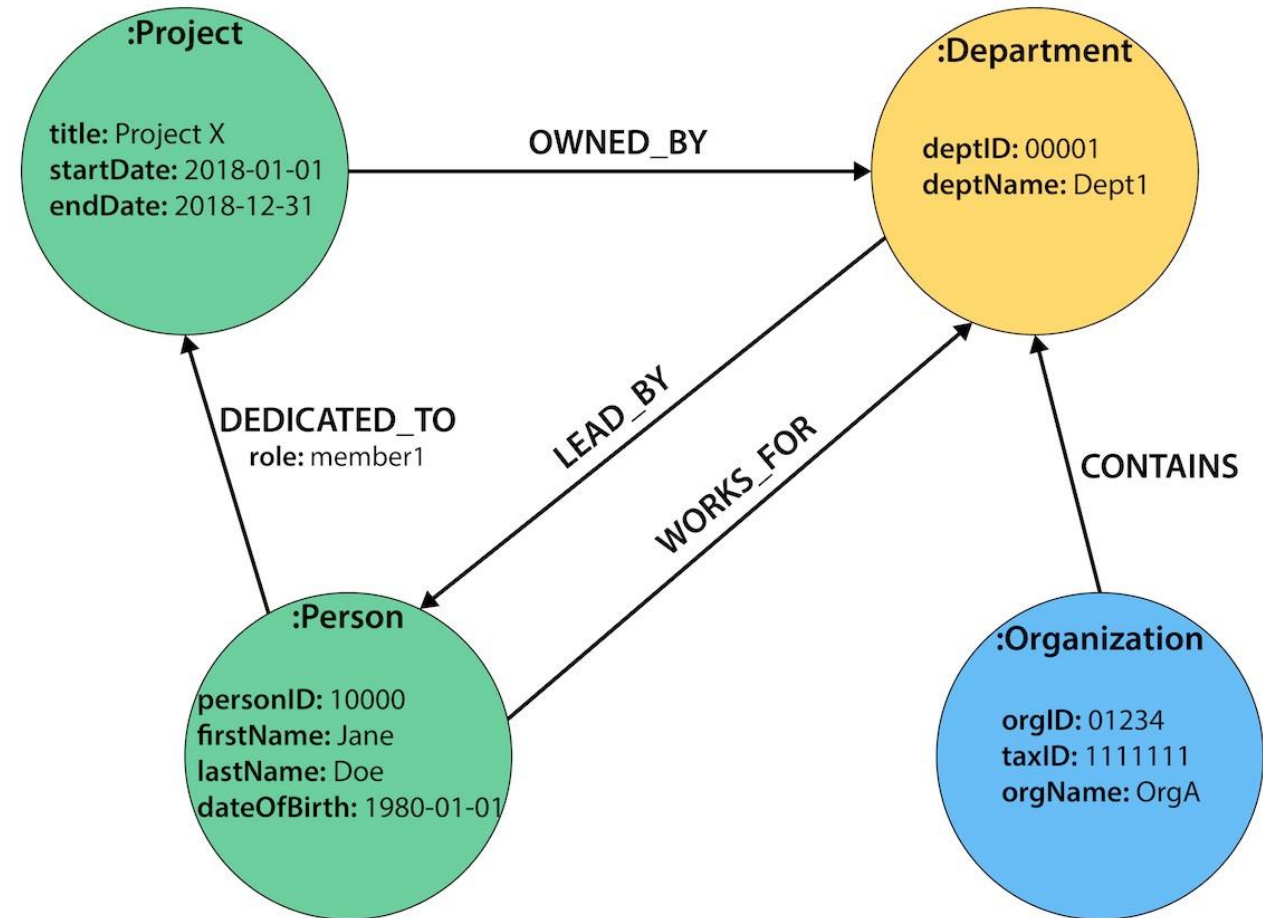
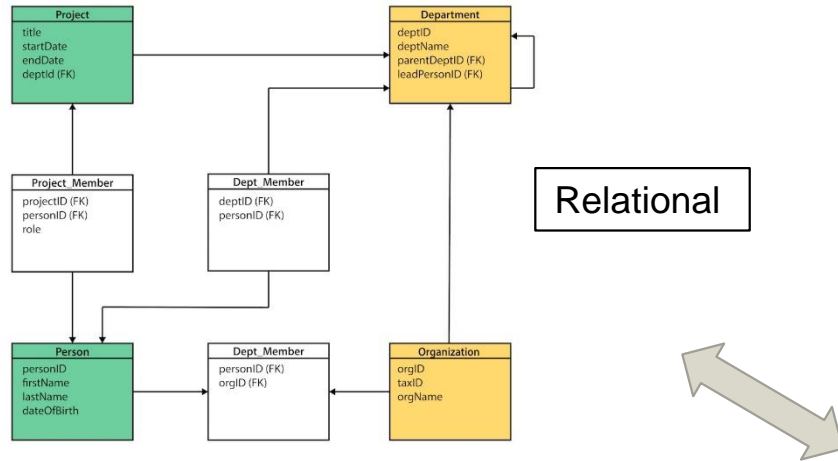


- Materializing of relationships at creation time, resulting in no penalties for complex runtime queries
- Constant time traversals for relationships in the graph both in depth and in breadth due to efficient representation of nodes and relationships
- All relationships are equally important, making it possible to materialize and use new relationships later on to “shortcut” and speed up the domain data when new needs arise
- Compact storage and memory caching for graphs, resulting in efficient scale-up and billions of nodes in one database on moderate hardware



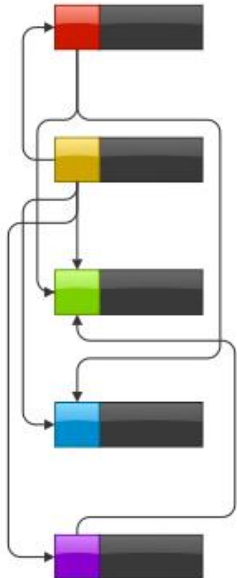
# Graph databases

## From relational to graph data model (2)





Graph



## 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)<-[:works_at]-(d:Department) WHERE
d.name = "IT Department" RETURN p.name
```

## Things to be wary about

- Data is generally duplicated – de-normalized data (data inconsistency?)
  - No standardized schema (by design)
  - No standard format for queries
  - No standard query language (NoSQL?)
  - How to handle complex data structures?
  - Application layer typically needs to enforce data integrity
  - No guarantee for support
- 
- CAP: Consistency, Availability, Partition Tolerance can't be achieved at the same time!



3.1. Introduction to distributed systems and middleware

3.2. Database-centric architectures

3.2.1. Data warehousing and business intelligence

3.2.2. Big data architectures

3.2.2.1. Scalability

3.2.2.2. NoSQL data stores

**3.2.2.3. Data processing**

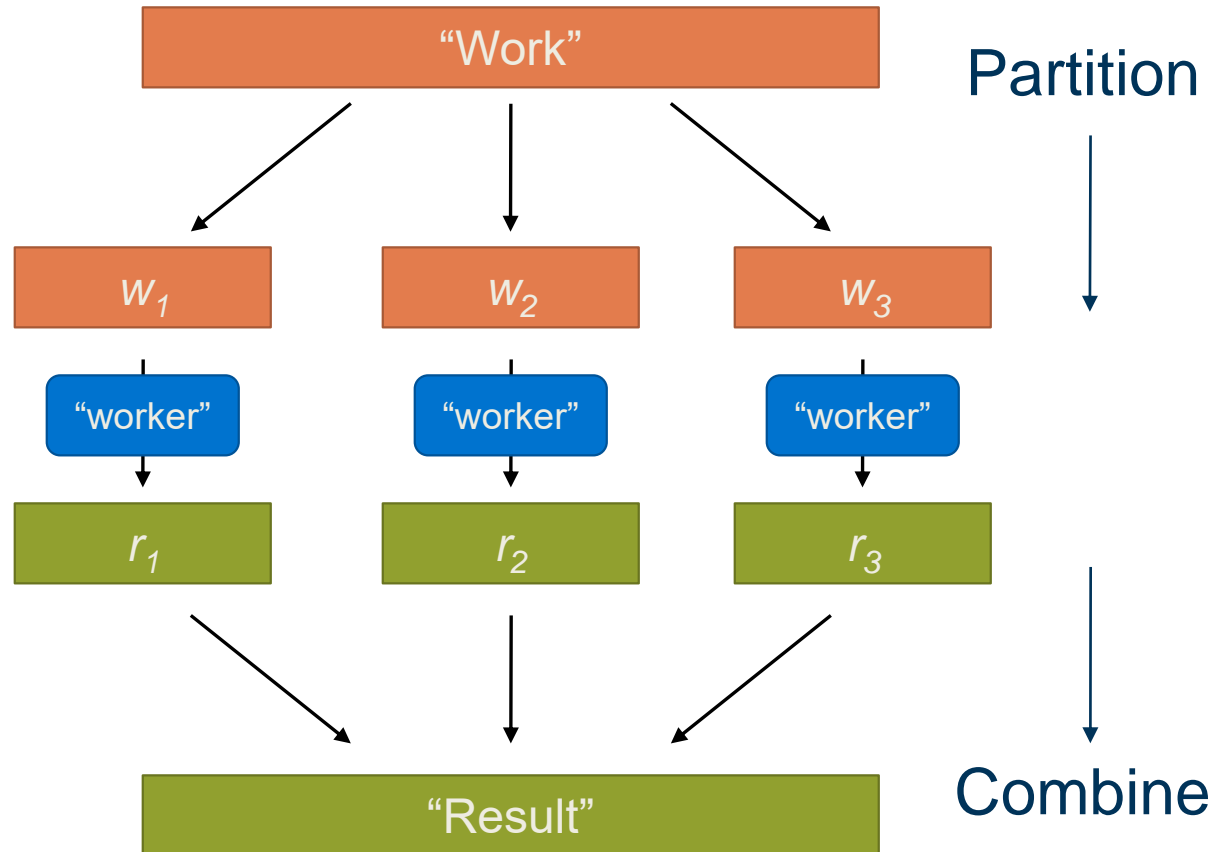
3.3. Message-oriented architectures

3.4. Object-oriented architectures

3.5. Component-based architectures

3.6. Service-oriented architectures

3.7. Blockchain-based systems



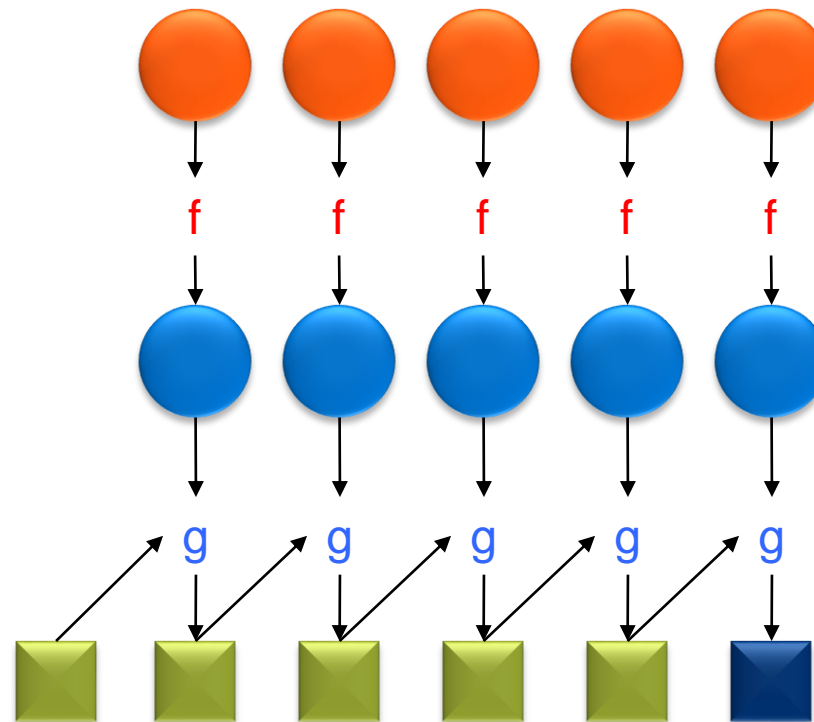
- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

- **Map** Iterate over a large number of records
  - Extract something of interest from each
  - Shuffle and sort intermediate results
  - Aggregate intermediate results
  - Generate final output
- **Reduce**

Key idea: provide a functional abstraction for these two operations

Map

Fold

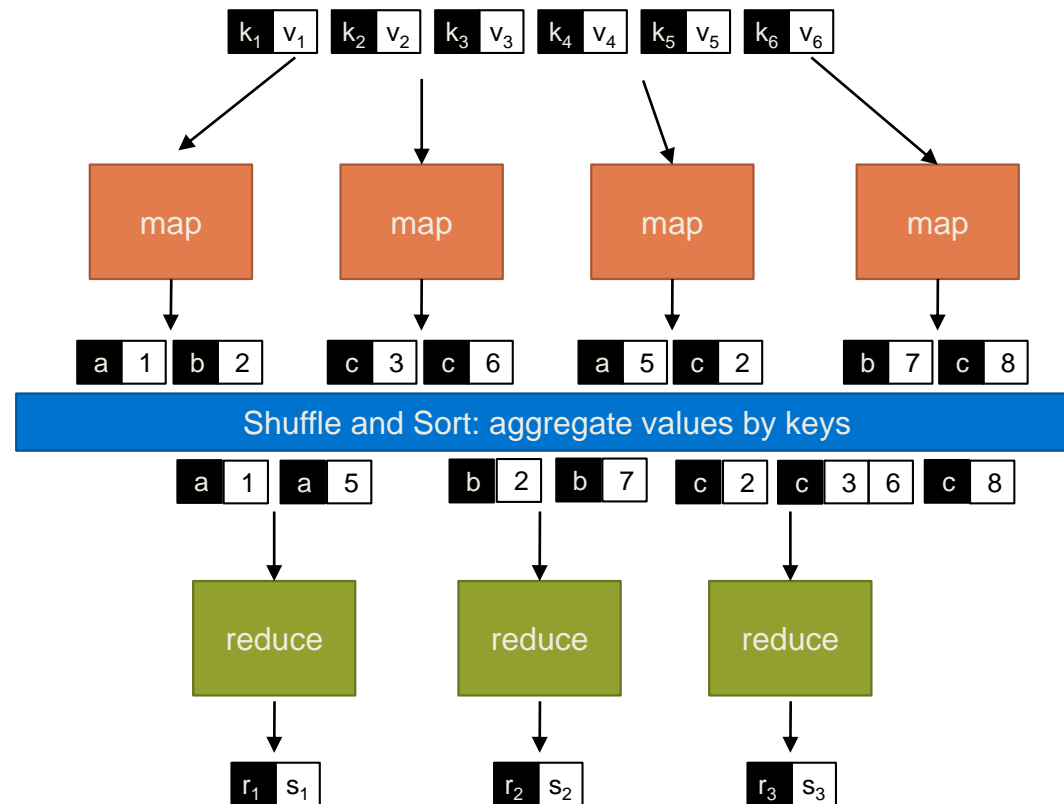


# Map and reduce functions

- Programmers specify two functions:
  - map**  $(k, v) \rightarrow [(k', v')]$
  - reduce**  $(k', [v']) \rightarrow [(k', v')]$
  - All values with the same key are sent to the same reducer

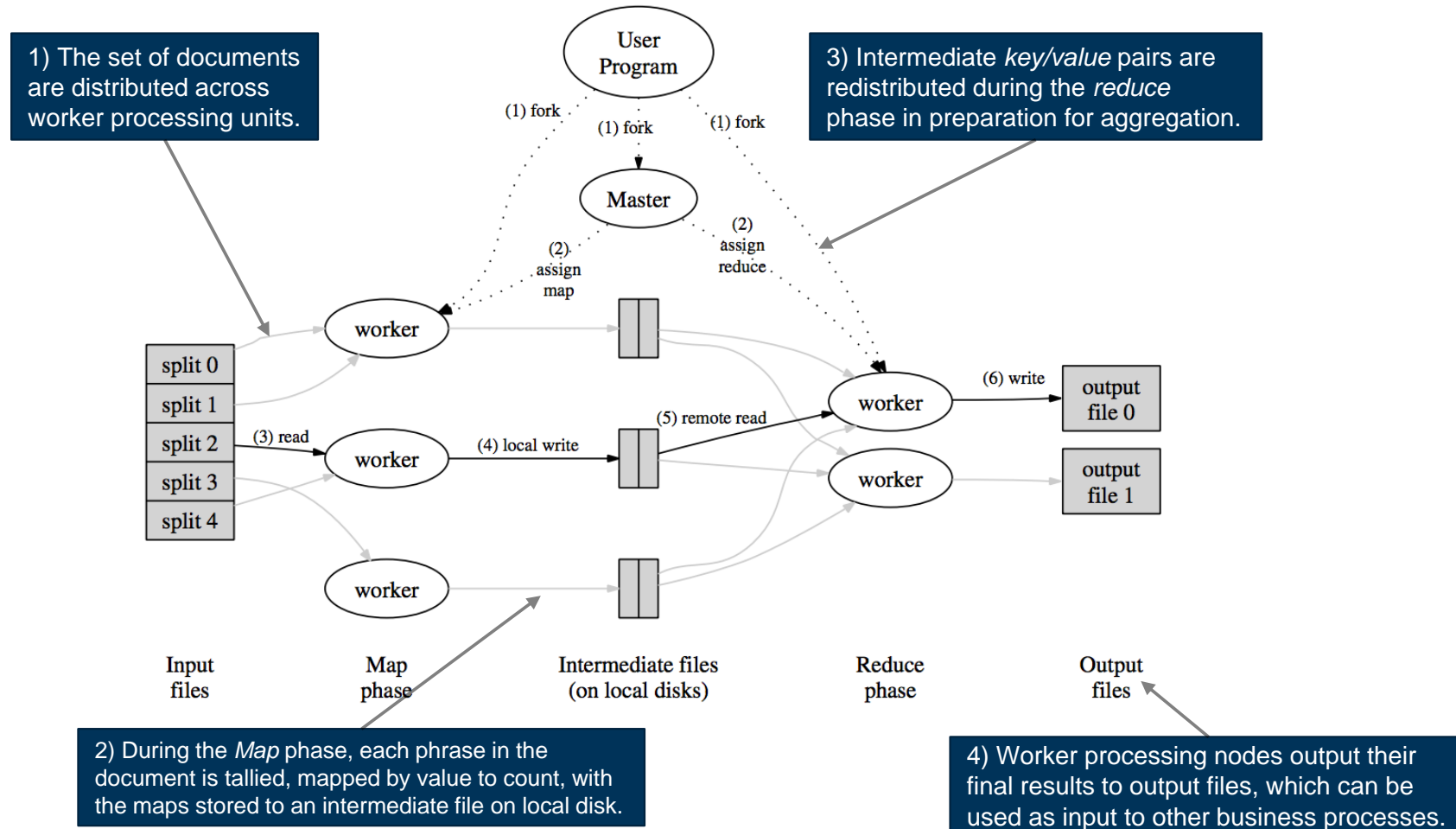
Execution framework handles

- Scheduling
- Data distribution
- Synchronization
- Errors and faults



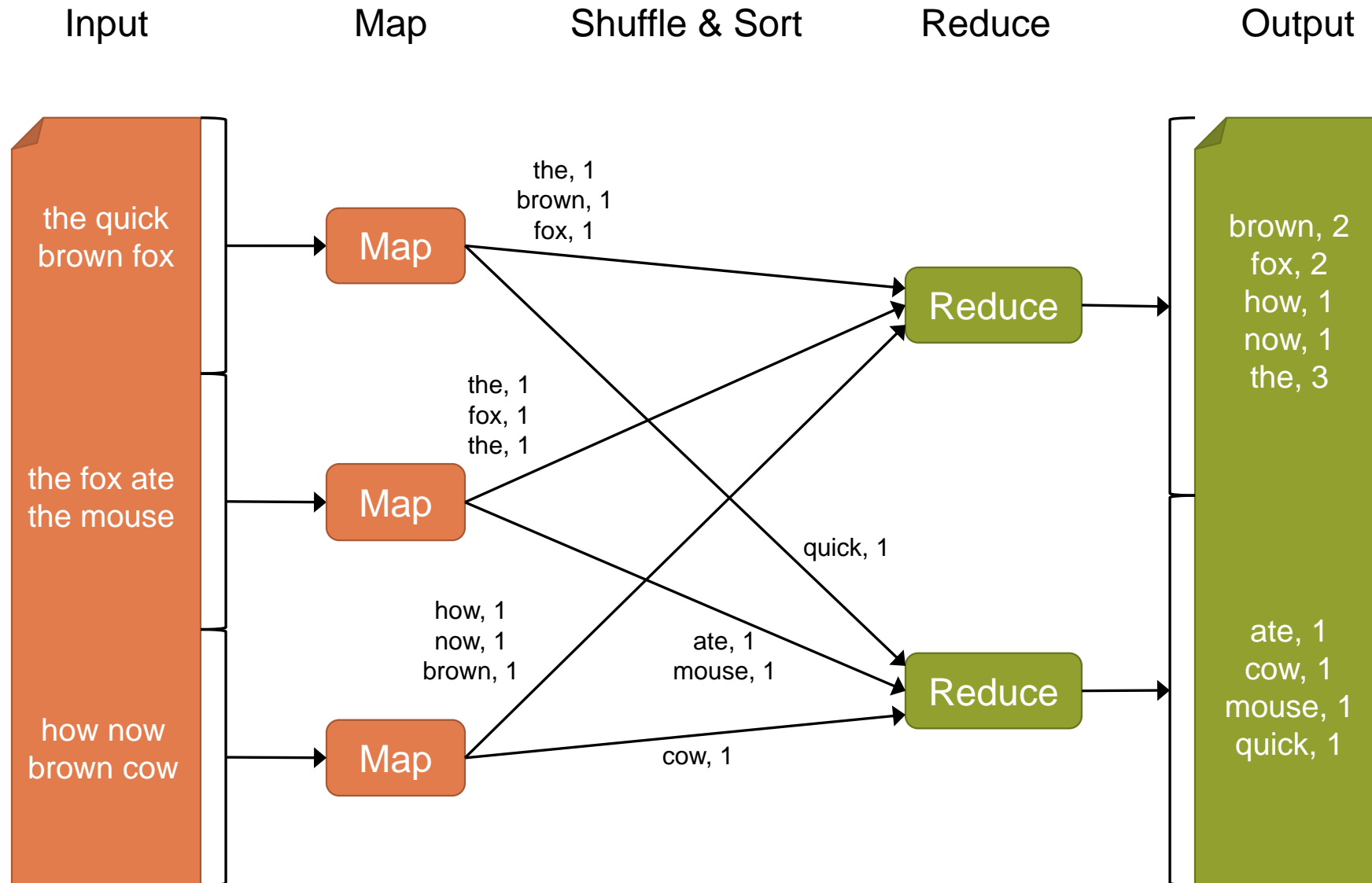
### Splitting and shuffling

### Computation and reduction



["MapReduce: simplified data processing on large clusters." J. Dean and S. Ghemawat (2008)]

```
map(String key, String value):  
    // key: document name  
    // value: document contents  
    for each word w in value:  
        EmitIntermediate(w, "1");  
  
reduce(String key, Iterator values):  
    // key: a word  
    // values: a list of counts  
    int result = 0;  
    for each v in values:  
        result += ParseInt(v);  
    Emit(AsString(result));
```





Applied with success in a wide area of “Big data” jobs

**Distributed grep:** The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

**Count of URL access frequency:** The map function processes logs of web page requests and outputs {URL, 1}. The reduce function adds all values for the same URL and emits a {URL, total count} pair.

**Reverse web-link graph:** The map function outputs {target, source} pairs for each link to a target URL found in a page named source. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: {target, list(source)}

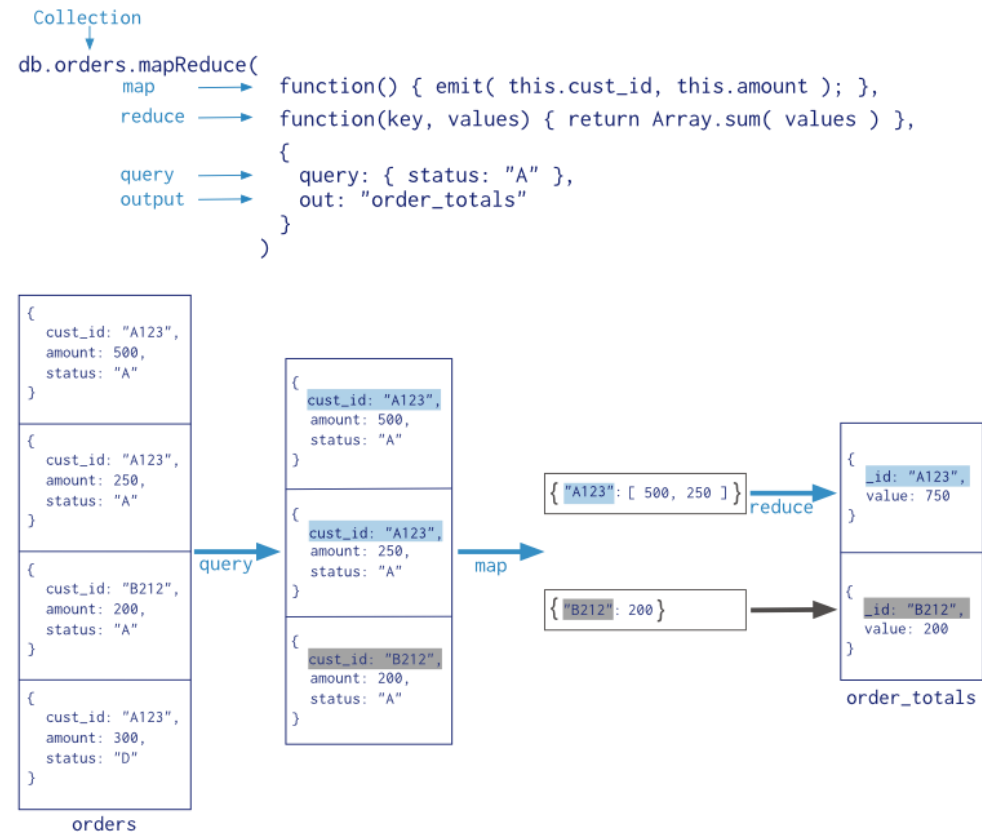
**Inverted index:** The map function parses each document, and emits a sequence of {word, document ID} pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a {word, list(document ID)} pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

# Evolution of Big Data query processing

- In 2021, it is no longer necessary for application programmers to write distributed map-reduce algorithms.
- Instead, they can use SQL queries provided by BigData database systems which optimize and perform the necessary distributed data access.
- The expressive power of these SQL dialects is growing, but still limited.
- There are competing open source and commercial solutions, see, e.g. <https://rakam.io/blog/6-sql-data-warehouse-solutions-for-big-data-analysts-with-their-pros-and-cons-18d34ca58fa6/>

# Example: MapReduce in MongoDB

Many document database have advanced big data features and are highly fitted for scaling scenarios. MongoDB allows map-reduce jobs and aggregation operations.



# Software architectures and their trade-offs

3.1. Introduction to distributed systems and middleware

3.2. Database-centric architectures

## **3.3. Message-oriented architectures**

3.4. Object-oriented architectures

3.5. Component-based architectures

3.6. Service-oriented architectures

3.7. Blockchain-based systems