A distributed system is one where the failure of some computer I've never heard of can keep me from getting my work done.

— Leslie Lamport

6

On Distributed Data Management

The discrepancy between the growth of data volume and the growth of computational power limits the possibilities of vertical scaling (i.e., adding more power to a machine). This leaves two possibilities for dealing with growing amounts of data: by developing more efficient algorithms or by horizontally scaling the system to multiple machines—or a combination of both approaches, which is what we propose in this thesis. While the data models presented in Chapter 4 introduced the foundations for the first approach, in this chapter, we introduce the foundations for the second approach.

Add supporting reference

Horizontal scaling means scaling by adding more nodes (i.e., computers, servers). This allows to parallelize the processing of workloads. However, distribution also introduces several new challenges. In the context of this thesis, there are two important topics: Firstly, the *management of resources* in such a distributed system. This includes the selection of distribution models (→ Section 6.1) and the efficient allocation of data to stores (→ Section 6.2). Secondly, ensuring the *consistency* of the data using distributed transactions (→ Section 6.3) and dealing with data freshness (→ Section 6.4).

A major and also extensively studied problem in the context of distributed database systems is *concurrency control* [OV11]. However, due to the proposed architecture model of the PolyDBMS, concurrency control is identical to a non-distributed database system. All queries go through the PolyDBMS, acting as a central instance, which allows handling concurrency control solely in this central system (see discussion of SS2PL in **??**). Distributed concurrency control is therefore not discussed in this chapter.

Add

Another major topic with distributed systems is the handling of failing nodes or network connections. The *CAP theorem* [GL02] introduced by Eric Brewer states, that in such an event, it is not possible to keep the system available and at the same time maintain the consistency of the data. The fundamental problem behind this is, that a

node in a network cannot distinguish whether another node has failed or only the network link between the nodes (called a network partitioning). This is also the case for a PolyDBMS relaying on underlying storage engines deployed on different machines. However, due to the architecture of a PolyDBMS where queries are only accepted by a central instance, there is no immediate risk of inconsistencies. This would only be the case if other systems would directly interact with the underlying storage engines. Nevertheless, strategies for handling failing storage systems in a PolyDBMS or even having multiple instances of the PolyDBMS itself would be very interesting aspects. However, this aspect will not be considered further in the context of this thesis.

6.1 Distribution Models

In this thesis, we distinguish between *partitioning* and *allocation*. Partitioning is the process of dividing a schema object into multiple partitions. These partitions can be of different sizes. Allocation is the process of assigning partitions to physical storage nodes. If the same partition is allocated to multiple nodes, this is called *replication*. This is depicted in Figure 6.1. A schema object (e.g., a relation) is divided in five logical partitions. These partitions are allocated to three nodes (i.e., database systems). Except for the partition P1, all partitions are allocated twice; thus, they are replicated. Partition P1 is only stored on Node A and is therefore not replicated. However, replication does not require partitioning a schema object into multiple partitions. It is also possible to assign the whole schema object to multiple nodes.

It can be distinguished between two fundamental forms of partitioning [OV11]: *horizontal partitioning* and *vertical partitioning*. Both forms of partitioning split the data of a schema object into multiple parts. However, they differ in how the data is divided and how the schema of the partitions looks like. In literature, partitioning is oft also called fragmentation [Dad96].

In the **horizontal partitioning**, a schema object is divided along its data. Each partition therefore consists of the full schema of the schema object (e.g., all attributes of a relation) but only a subset of the data (e.g., the tuples of a relation). The mapping of data items to partition can either be specified explicitly (e.g., by lists of values for every partition) or is done based on a round-robin or hash based approach.

The **vertical partitioning** splits the schema of a schema object into partitions containing a subset of the schema (e.g., a subset of the attributes of a relation) and only the data belonging to this subset of the schema (e.g., the data for these attributes of the relation).

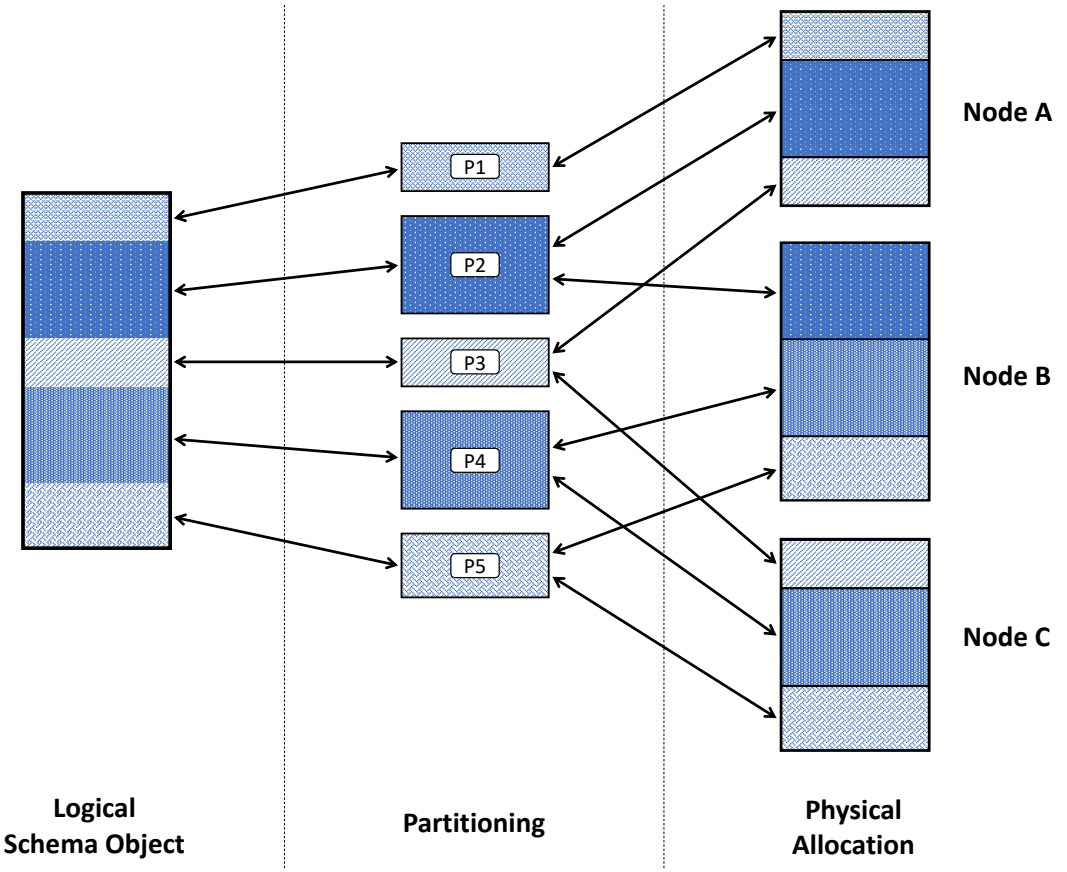

Figure 6.1 Data partitioning and allocation (based on [Dad96]).

In order to recombine the vertical partitions, there needs to be some redundantly stored data (e.g., the primary key of a relation) that uniquely identifies the individual data items of the partitions (e.g., the individual tuples of a relation).

It is also possible to combine vertical and horizontal partitioning. In literature, this is usually called *hybrid partitioning* [Dad96]. Furthermore, both forms of partitioning can be combined with replication [OV11].

The **allocation problem** refers to the issue of finding the optimal distribution of partitions to nodes [OV11]. This heavily depends on the data and the workload. However, based on the exhaustive analysis on the impact of data replication and data partitioning on the query performance presented in [NJ00], we can draw some general conclusions.

The performance of data modification queries is usually negatively affected by data replication. This makes sense since changes need to be applied to all replicas. With partitioning, it depends on the workload: if queries only modify individual partitions, it can have a positive impact on the performance of data modification queries.

Read-only queries typically benefit from data replication. The same applies for partitioning if only one or very view partitions are queried. If several partitions need to be accessed, this has a negative impact on the performance of the query. In general, it can be concluded that horizontal partitioning is beneficial for OLTP and bad for OLAP workloads.

6.2 Temperature-aware Data Management

The storage device has a major impact on the performance of a database system [OH11]. Traditionally, database systems are build based on a *two-layer architecture* [Hä05] with HDDs as persistence storage layer and the main memory as buffer and cache. In recent years, HDDs got more and more replaced by *solid-state drives* (SSDs). While SDDs provide a higher performance, they are also significantly more expensive. Together with even faster and more expensive storage technologies like *non-volatile random-access memory* (NVRAM) and very slow but also cheap archival storage systems, database administrators can choose from a wide range of storage technologies—depending on their budget.

However, there is a problem: as outlined by the data backup service provider BackBlaze in a blog post [Kle17], the amount of data is growing much faster than the cost of storage devices is decreasing. As a consequence, even maintaining current performance is becoming more and more expensive due to the constant growth of data. *Temperature-aware data management* is a technique to increase the overall performance, while at the same time, reducing storage costs. It makes use of the fact that in many scenarios, only a fraction of the data is heavily used.

Research conducted by the database company Teradata has shown [Gra12] that 85% of the I/O operations use the same 10% of data. While this is of course only an average that depends on the use-case, the data has been gathered from productive systems and shows a huge potential for optimization.

In the data management community it is common practice to describe the access frequency of a certain data set by a *temperature* [GP87]. Data that is currently being accessed very frequently is referred to as *hot*, while data that has not been accessed for a long time is considered *cold*. In between, there can be various shades of *warm*. Temperature-aware data management refers to the approach of storing hot data on fast (and expensive) storage, warm data on somewhat slower storage, and cold data on slow but also very cheap storage.

There are various approaches for calculating the temperature of a data item. The authors of [LLS13] suggest logging the accesses to a data item and performing an offline analysis to estimate the access frequency. In [PD11], techniques for identifying hot and cold data using multiple Bloom filters and in [PDN+12] using sampling-based techniques are presented. Hot and cold data identification is also a topic of extensive discussion in the context of flash memory devices. The authors of [HKC06] present an online approach to hot data identification using multiple hash functions. Their work also includes an analytical study of the probability of false identifications.

6.3 Distributed Transactions

A distributed transaction is a transaction that involves two or more participants. These transactions also need to provide the ACID properties outlined in **??**. The correct completion of distributed transactions is ensured using *consensus algorithms*. A major consensus algorithm widely used in the context of distributed database systems is the *two-phase-commit* (2PC) protocol. Other well known consensus algorithms are *RAFT* [OO14] and *Paxos* [Lam98]. In this thesis, we focus on the 2PC algorithm.

Add

The 2PC protocol as described in [WV02] models transactions over multiple database systems which can be deployed on different nodes of a network. There are two roles in the 2PC protocol: the *coordinator* and the *participants*. The coordinator role is typically taken by the node where the transaction has been initialized **??**. This coordinator is responsible for steering the correct execution of the transaction. All database systems involved in the transaction are called participants. The possible states and transitions of the coordinator and the participants are depicted in Figure 6.2.

As the name implies, the 2PC protocol consists of two phases: the *voting phase* and the *commit phase*. In the voting phase, the coordinator requests all participants to "prepare". If the participants are able to commit, they vote with "yes". Otherwise, the vote with "no". In the commit phase, the coordinator decides based on the votes whether to commit or to abort the transaction. However, the transaction can only be committed if all participants have voted with "yes". The coordinator informs the participants about its decision. The participants then execute this decision. Since the participants already confirmed in the previous phase that they are able to commit, it is guaranteed that the commit happens on all nodes involved in the transactions.

Without such a commit protocol, it could happen that a node is unable to commit a transaction (e.g., because it violates a constraint). This would lead to inconsistencies

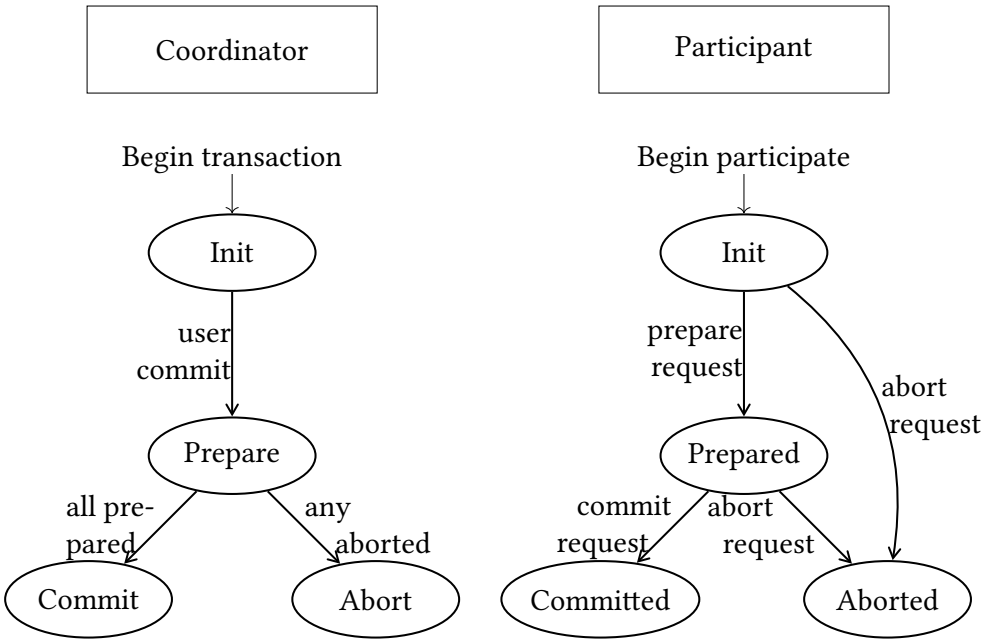

Figure 6.2 Possible states of the coordinator and the participants in the two-phase-commit protocol (based on [Wan21]).

since changes would be committed on some nodes but not all nodes; a violation of the *atomicity* requirement of ACID.

6.4 Data Freshness

If data is replicated in a distributed system, it can be distinguished between two forms of *update propagation*: *eager* and *lazy*. These describe when changes to a data item get applied to the nodes where a copy of the partition is physically allocated.

In systems where updates get propagated **eagerly**, all copies are modified immediately. Subsequent queries can therefore read from any of the nodes and always get the latest data. However, there are also techniques which defer the propagation to the commit time instead of applying the change immediately on all nodes [OV11]. In this thesis, we refer to eager propagation as synchronously applying changes to all nodes. The main disadvantage of eager update propagation are the usually higher response times of an update query since the query can not complete until all nodes have been updated.

Add supporting reference

If updates get propagated **lazily**, changes are not necessarily performed immediately or even within the context of the transaction. The propagation of updates is done asynchronously from the original transaction [OV11]. Subsequent queries therefore might read outdated data. The main advantage of lazy update techniques is the usually dras-

tically lower response time since only one node needs to be update for the query to complete.

Add supporting reference

A consequence of systems applying a lazy update propagation strategy is, that the distributed system might stores multiple versions of the same data item. Nodes containing such outdated versions of a data items are sad to have a lower *data freshness*. However, as outlined in [VSB+10] such slightly outdated data is acceptable for several applications. The authors argue that this can be exploited by keeping replicas at various levels of freshness and executing queries accepting this level of freshness on such outdated, not yet-to-be updated partitions. Such a combination of eager and lazy replication offers a great trade off between the efficient usage of available resources and the response time of update queries.

There is no commonly agreed definition of **data freshness**. In [NLF99] and [Red96] it is described as the percentage of unchanged data items. According to [CGM00], the freshness of a element e_i at a time t can be expressed as:

$$F(e_i, t) := \begin{cases} 1 & \text{if } e_i \text{ is up-to-date at time } t \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

The average freshness of a schema object s at a time t is given by:

$$F(s, t) := \frac{1}{N} \sum_{i=1}^{N} F(e_i, t) \tag{6.2}$$

The authors of [CGM00] also introduce the notion of *age*, capturing the average time since the outdated items have been updated the last time. The age of a data item e_i at time t is given as:

$$A(e_i, t) := \begin{cases} 0 & \text{if } e_i \text{ is up-to-date at time } t \\ (t - \text{modification time of } e_i) & \text{otherwise} \end{cases} \tag{6.3}$$

With this, the average freshness of a schema object s at a time t is given by:

$$A(s, t) := \frac{1}{N} \sum_{i=1}^{N} F(e_i, t) \tag{6.4}$$

Another metric for measuring freshness is the number of updates a data item or schema object is behind the latest version. By [BP04], this is called *obsolescence metric*. The authors also introduce a *timeliness metric*, that takes the update frequency of the schema object into account. Instead of calculating the actual "outdatedness", it is estimated based on the time elapsed since the last update of the node in relation to the update frequency of the schema object.