Chapter 2
Related Work and Background

This chapter presents the background as well as the related work for this work. Instead of separating the chapters *related work* and *background*, both topics are presented together in one chapter, giving the reader the advantage of understanding underlying concepts and getting to know the respective related work in one stroke. Each section and subsection is preceded by a short summary. Additional related work is presented where appropriate for example Chap. 6 starts with discussing system models which are related to the system model presented in remainder of this chapter, while Chap. 7 contains an overview on state-of-the-art distributed join algorithms. The four major areas which influence this work are current computing hardware trends, in-memory database management systems, parallel database management systems, as well as cloud storage systems. The subsequent sections and subsections also include discussions regarding how the different areas influence each other.

2.1 Current Computing Hardware Trends

Summary: This section introduces the computing hardware trends by describing the foundation for current computer architecture.

John Von Neumann described in 1945 [vN93] a computer architecture that consists of the following basic components: (a) an Arithmetic Logic Unit (ALU), also known as processor which executes calculations and logical operations; (b) memory which holds a program and its to be processed data; (c) a Control Unit that moves program and data into and out of the memory and executes the program instructions. For supporting the execution of the instructions the Control Unit can use a Register for storing intermediate values; (d) an Input and Output mechanism allows the interaction with external entities such as a user via keyboard and screen. The original Von Neumann architecture included the interconnection of the different architectural components, however an explicit System Bus has been added later to be able to connect a non-volatile memory medium for persistent data storage [NL06]. Figure 2.1 depicts the Von Neumann Architecture with an added system bus.

C. Tinnefeld, *Building a Columnar Database on RAMCloud*, In-Memory Data Management Research, DOI 10.1007/978-3-319-20711-7_2

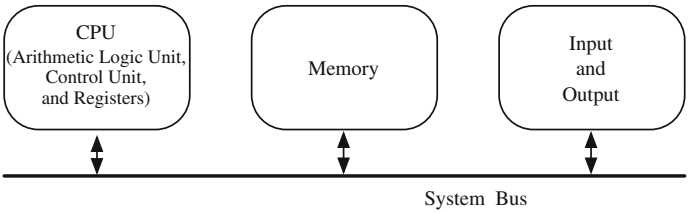

Fig. 2.1 Modified Von Neumann Architecture with added system bus

As indicated in the description of the Von Neumann architecture, there are different types of memory technologies which vary in properties such as cost ($ per bit), access performance, and volatility. Inside a single computer, a combination of different memory technologies is used with the intention of combining their advantages. The combination of the different technologies is done by aligning them in a hierarchy in order to create the illusion that the overall memory is as large as the largest level of the hierarchy. Such a memory hierarchy can be composed of multiple levels and data is transferred between two adjacent levels at a time. As shown in Fig. 2.2, the upper levels are usually smaller, faster, more expensive, and closer to the CPU, whereas the lower levels provide more capacity (due to the lower costs), but are slower in terms of access performance. The textbook memory technology examples from Patterson and Hennessy [PH08] are SRAM (static random access memory), DRAM (dynamic random access memory), and magnetic disk which build a three-level hierarchy: SRAM provides fast access times, but requires more transistors for storing a single bit which makes it expensive and taking up more space [HP11]. SRAM is used in today's computers as a cache very close to the CPU and is the first memory level in this example. DRAM is less costly per bit than SRAM, but also substantially slower and is the second memory level and is used as main memory for holding currently processed data and programs. In addition, DRAM cells need

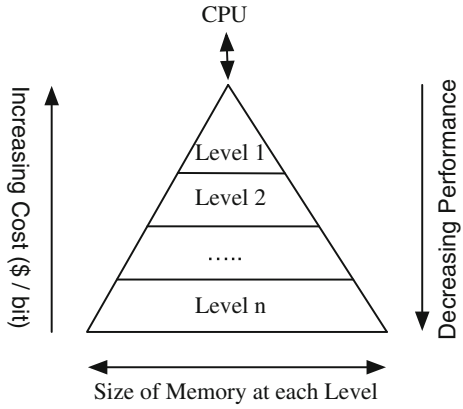

Fig. 2.2 Memory hierarchy as described by Patterson and Hennessy

constant energy supply otherwise they loose the stored information which makes them volatile. In order to permanently store data, a third memory hierarchy is introduced that uses a magnetic disk. However, the moving parts inside a magnetic disk result in a penalty with regards to the access performance in comparison to DRAM. Two other frequently used terms for differentiating between volatile and non-volatile memory are primary and secondary memory [PH08]: primary memory is a synonym for volatile memory such as main memory holding currently processed programs and data, whereas secondary memory is non-volatile memory storing programs and data between runs. A shortcoming of the Von Neumann Architecture is that the CPU can either read an instruction or data from memory at the same time. This is addressed by the Harvard architecture [HP11] which physically separates storage and signal pathways for instructions and data.

Reflecting on the Von Neumann Architecture and the aspect of memory hierarchies, it becomes apparent that a computer is a combination of different hardware components with unique properties and that the capabilities of those hardware components ultimately set the boundaries of the capabilities of the to-be-run programs on that computer. Consequently, hardware trends also impact the way computer programs are designed and utilize the underlying hardware. In the remainder of this section we want to describe three hardware trends which contribute to the motivation for this work.

2.1.1 Larger and Cheaper Main Memory Capacities

Summary: This subsection quantifies the advancements in main memory capacities and the price reduction over the last 18 years.

The price for main memory DRAM modules has dropped constantly during the previous years. As shown in Fig. 2.3, the price for one megabyte of main memory

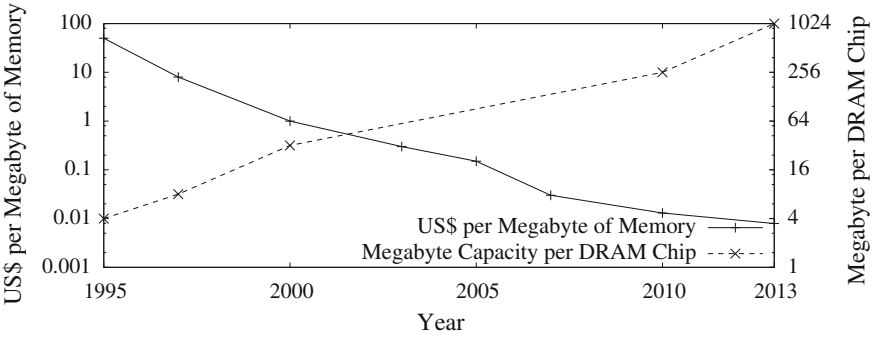

Fig. 2.3 Evolution of memory price development and capacity per chip advancement [Pla11a, PH08]. Please note that both y-axes have a logarithmic scale

used to be $50 in 1995, and dropped to $0.03 in 2007—a price reduction of three orders of magnitudes in about 20 years. In addition, chip manufacturers managed to pack transistors and capacitors more densely, which increased the capacity per chip. In 1995, a single DRAM chip had the capacity of 4 Megabytes, which increased to 1024 Megabyte per chip in 2013. Putting 32 of those chips on a single DRAM module gives it a capacity of 32 Gigabytes. Nowadays, server mainboards (e.g. Intel[1] Server Board S4600LH2 [Int13]) can hold up to 48 of those modules, resulting in a capacity of 1.5 terabyte of main memory per board. Such an Intel server board equipped with 1.5 terabyte of Kingston server-grade memory can be bought for under $25,000 (undiscounted retail price [new13]), which illustrates the combination of price decline and advancement in capacity per chip.

Despite the previously described developments, solid-state drives (SSD) and hard-disk drives (HDD) still have a more attractive price point—e.g. the cost per megabyte capacity of a three terabyte hard-disk is about $0.004 [new13]. In addition, DRAM is a volatile storage and as long as non-volatile memory is not being mass-produced, one always needs the same capacity of SSD/HDD storage somewhere to durably store the data being kept and processed in DRAM (the situation is comparable to alternative energy sources—you still need the coal power plant when the sun is not shining or the wind is not blowing). However, in the end the performance advantage of operating on main memory outweighs the higher cost per megabyte in the context of performance critical applications such as large-scale web applications [mem13a] or in-memory databases [Pla11a].

2.1.2 Multi-Core Processors and the Memory Wall

Summary: This subsection describes the development of having multiple cores per processor and the resulting main memory access bottleneck.

In 1965, Gordon Moore made a statement about the future development of the complexity of integrated circuits in the semiconductor industry [Moo65]. His prediction that the number of transistors on a single chip is doubled approximately every two years became famous as Moore's Law, as it turned out to be a relatively accurate prediction.

The development of the processors clock rate and the number of processor cores is of relevance: as depicted in Fig. 2.4a, Intel CPUs reached a clock rate of 3000 Mhz in 2003. Until then the clock rate improved from yearly which used to be convenient for a programmer, as he usually did not have to adjust his code in order to leverage the capabilities of a new generation of processors. This changed for Intel processors between 2003 and 2006, when a further increasement of the clock rate would have

[1] There is a great variety of computer processor and mainboard vendors such as AMD [AMD13] , GIGABYTE [GIG13] or Intel [Int13]. For the sake of better comparability throughout the different subsections, this section cites only Intel processor and mainboard products. In addition, Intel holds a market share of over 90 % in the worldwide server processor market in 2012 [RS13].

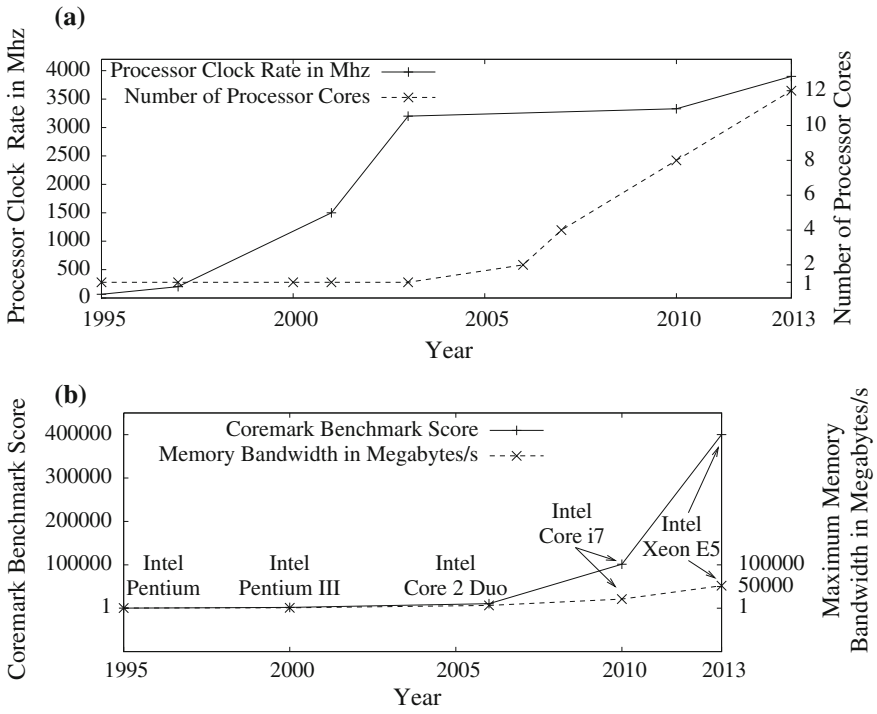

Fig. 2.4 Illustration of the evolution of Intel processors from 1995 until 2013. **a** Evolution of Intel processor clock rate and number of processor cores [Int13]. The figure shows that Intel distributed a processor with more than 3000 Mhz clock rate (Intel Pentium 4 Xeon) already in 2003, but until 2013 the clock rate only evolved to 3900 Mhz (Intel Core i7 Extreme Edition). The first mass-produced processor from Intel with two cores (Intel Pentium Dual-Core) came out in 2006: the number of cores per processor evolved to twelve in 2013 (Intel Xeon E5). **b** Comparison of Coremark Benchmark Score [The13] and memory bandwidth (maximum specified rate at which data can be read from or stored to a semiconductor memory by the processor) of selected Intel processors [Int13] over time. The Coremark benchmark measures the performance of a CPU by executing algorithms such as list processing or matrix manipulation and intends to replace the Dhrystone [Wei84] benchmark. The figure illustrates that the processing power of CPUs increased more significantly (by introducing multiple physical processing cores per CPU) than its memory bandwidth

resulted in too much power consumption and heat emission. Instead, the clock rate remained relatively stable, but having more than one physical processing core per (multi-core) CPU was introduced. This led to the saying that the *free lunch is over* [Sut05], now application developers had to write their software accordingly to utilize the capabilities of multi-core CPUs. This is also expressed by Amdahl's Law, which says that the speed up of a system can also be defined as the fraction of code that can be parallelized [Amd67]. Reflecting on the importance of main memory as mentioned in the previous subsection, Fig. 2.4b shows the development of CPU processing power and its memory bandwidth. One can observe that with the advent of multiple cores

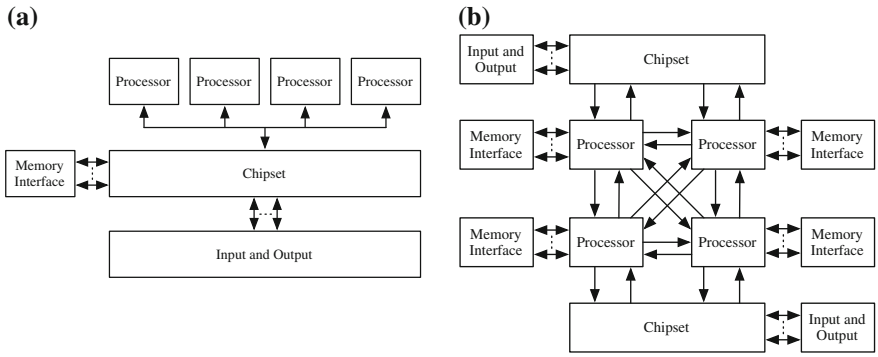

Fig. 2.5 Two different processor interconnect architectures. **a** Shared Front-Side Bus. **b** Intel's Quick Path Interconnect

per processor, processing power spiked significantly: an equivalent increase in the maximum specified rate at which data can be read from or stored into a semiconductor memory by the processor could not be realized [BGK96].

As described in the beginning of this section, a processor is not directly wired to the main memory, but accesses it over a system bus. This system bus becomes a bottle neck when multiple processors or processor cores utilize it. Figure 2.5a shows a shared front-side bus architecture where several processors access the memory over a shared bus. Here, the access time to data in memory is independent regardless which processor makes the request or which memory chip contains the transferred data: this is called uniform memory access (UMA). In order to overcome the bottleneck of having a single shared bus, for example, Intel introduced Quick Path Interconnect as depicted in Fig. 2.5b, where every processor has its exclusively assigned memory. In addition, each processor is directly interconnected with each other, which increases the overall bandwidth between the different processors and the memory. A single processor can access its local memory or the memory of another processor, whereat the memory access time depends on the memory location relative to the processor. Therefore, such an architecture is described as NUMA, standing for non-uniform memory access.

2.1.3 Switch Fabric Network and Remote Direct Memory Access

Summary: This subsection quantifies the advancements of network bandwidth and latency over the last 18 years, as well as the closing performance gap between accessing main memory inside a server and that of a remote server.

The previous subsection describes the performance characteristics when operating on the main memory inside a single server. Although Sect. 2.1.1 emphasizes the

growing main memory capacities inside a single server, the storage space requirements from an application can exceed this capacity. When utilizing the main memory capacities from several servers, the performance characteristics from the network interconnect between the servers have to be considered as well. Modern network technologies such as InfiniBand or Ethernet Fabrics have a switched fabric topology which means that (a) each network node connects with each other via one or more switches and (b) that the connection between two nodes is established based on the crossbar switch theory [Mat01] resulting in no resource conflicts with connections between any other nodes at the same time [GS02]: in the case of InfiniBand, this results in full bisection bandwidth between any two nodes at any time. In addition the InfiniBand specification [Ass13] describes that an InfiniBand link can be operated at five different data rates: 0.25 GBytes/s for single data rate (SDR), 0.5 GBytes/s for double data rate, 1 GBytes/s for quad data rate (QDR), 1.7 GBytes/s for fourteen data rate (FDR), and 3.125 GBytes/s for enhanced data rate (EDR). In addition, an InfiniBand connection between two devices can aggregate several links in units of four and twelve (typically denoted as 4x or 12x). For example, the aggregation of four quad data rate links results in 4xQDR with a specified data rate of 4 GBytes/s. These specifications describe the effective theoretical unidirectional throughput, meaning that the overall bandwidth between two hosts can be twice as high.

Figure 2.6a compares the bandwidth specifications of main memory and network technologies. The figure shows how the bandwidth performance gap narrows down from over an order of magnitude (1995: 267 MBytes/s Fast Page Mode DRAM versus 12.5 MBytes/s Fast Ethernet) to a factor of 1.3 (2010: 16 GBytes/s DDR3-2000 SDRAM versus 12.5 GBytes/s 100 Gigabit Ethernet) over a period of 15 years, and that today's local area network technology specifies a higher bandwidth than memory (2013: 24 GBytes/s DDR3-3000 SDRAM versus 37.5 GBytes/s InfiniBand Enhanced Data Rate (EDR) 12x). Figure 2.6b compares the latency specifications of main memory and network technologies. The figure depicts how the latency performance gap has been reduced from five orders of magnitude (2000: 0.052 µs Double Data Rate SDRAM versus 340 µs Gigabit Ethernet) to two orders of magnitude (2013: 0.037 µs DDR3 SDRAM vs 1 µs InfiniBand). The recent improvement in network latency is the result of applying a technique called remote direct memory access (RDMA). RDMA enables the network interface card to transfer data directly into the main memory which bypasses the operating system by eliminating the need to copy data into the data buffers in the operating system (which is also known as zero-copy networking)—which in turn also increases the available bandwidth. In addition, transferring data via RDMA can be done without invoking the CPU [Mel13]. RDMA was originally intended to be used in the context of high-performance computing and is currently implemented in networking hardware such as InfiniBand. However, three trends indicate that RDMA can become wide-spread in the context of standardized computer server hardware: From a specification perspective, RDMA over Converged Ethernet (RoCE) allows remote direct memory access over an Ethernet network and network interface cards supporting RoCE are available on the market. From an operating system perspective, for example Microsoft supports RDMA in Windows 8 and Windows Server 2012 [WW13]. From a hardware perspective, it is likely that

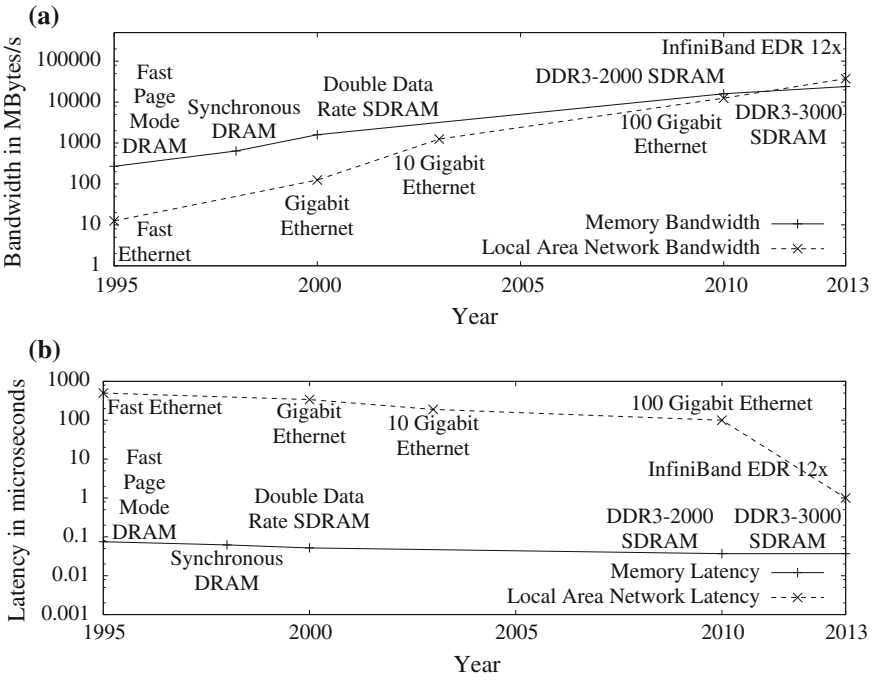

Fig. 2.6 Comparisons of memory and local area network bandwidth and latency specifications from 1995 until 2013. **a** Comparison of memory and local area network bandwidth specifications [HP11, Ass13]. The figure illustrates how the bandwidth performance gap narrows down from over an order of magnitude (1995: 267 MBytes/s Fast Page Mode DRAM vs. 12.5 MBytes/s Fast Ethernet) to a factor of 1.3 (2010: 16 GBytes/s DDR3-2000 SDRAM vs. 12.5 GBytes/s 100 Gigabit Ethernet) over a period of 15 years, and that today's local area network technology specifies a higher bandwidth than memory (2013: 24 GBytes/s DDR3-3000 SDRAM vs. 37.5 GBytes/s Enhanced Data Rate (EDR) 12x). **b** Comparison of memory and local area network latency specifications [HP11, Ass13]. The figure shows that the latency performance gap has been reduced from five orders of magnitude (2000: 0.052 µs Double Data Rate SDRAM vs. 340 µs Gigabit Ethernet) to one to two orders of magnitude (2013: 0.037 µs DDR3 SDRAM vs. 1 µs InfiniBand)

RDMA capable network controller chips become on-board commodity equipment on server-grade mainboards.

The comparison of main memory and network technology specifications suggest that the performance gap between operating on local (inside a single computer) and remote (between two separate computers) memory closes. Table 2.1 presents a comparison of hardware specifications and actual measurements in order to quantify the bandwidth and latency performance gap between main memory and network technologies. The Intel Nehalem architecture—which is going to be used in the context of the measurements—specifies a maximum bandwidth of 32 GBytes/s by combining its three memory channels per processor [Tho11]. The specified access latency for retrieving a single cache line from main memory that is not resident in the caches

Table 2.1 Bandwidth and latency comparison for accessing local (inside a single computer) and remote (between two separate computers) DRAM

	Specification		Measurements[3]	
	Intel Nehalem[Tho11] Specification	InfiniBand[Ass13] Specification 4xFDR/12xEDR	Intel Xeon Processor E5-4650	Mellanox ConnectX-3 4xFDR
Bandwidth	32 GBytes/s[1] Memory Bandwidth	6.75 GBytes/s[2] 37.5 GBytes/s[2]	10.2 GBytes/s[4] Memory Bandwidth	4.7 GBytes/s[6]
Latency	0.06 µs Uncached Main Memory Access	1 µs RDMA Operation End-to-End Latency	0.07 µs[5] Uncached Main Memory Access	1.87 µs[6] RDMA Operation End-to-End Latency

[1] Combined memory bandwidth of three memory channels per processor
[2] These are the specified actual data rates (not signaling rates) for 4xFDR respective 12xEDR
[3] Measurements have been performed on the following hardware: Intel Xeon E5-4650 CPU, 24 GB DDR3 DRAM, Mellanox ConnectX-3 MCX354A-FCBT 4xFDR InfiniBand NIC connected via Mellanox InfiniScale IV switch
[5] Measured via Bandwidth Benchmark from Z. Smith [Smil13]—Sequential read of 64MB sized objects
[4] Measured via Cache-Memory and TLB Calibration Tool from S. Manegold and S. Boncz [MDK02a]
[6] Measured via native *InfiniBand Open Fabrics Enterprise Distribution* benchmarking tools *ib_read_bw* and *ib_read_lat*

between the CPU and the main memory is 0.06 µs. Actual measurements with an Intel Xeon E5-4650 processor show a memory bandwidth of 10.2 GBytes/s and a memory access latency of 0.07 µs. The difference between the bandwidth specification and the measurement is the number of memory channels: the memory traversal of a data region executed by a single processor core usually invokes a single memory channel at a time, resulting in approximately one third of the specified bandwidth (due to the three available memory channels). Current typical InfiniBand equipment, such as Mellanox ConnectX-3 cards (e.g. Mellanox ConnectX-3 MCX354A-FCBT), support 4xFDR, resulting in a unidirectional data rate of 6.75 GBytes/s between two devises (as previously mentioned, the current InfiniBand specification [Ass13] itself supports up to 37.5 GBytes/s (12xEDR)). The end-to-end latency for a RDMA operation is specified with 1 µs. Measurements between two machines, each equipped with a Mellanox ConnectX-3 MCX354A-FCBT card and connected via a Mellanox InfiniScale IV switch, reveal a unidirectional bandwidth of 4.7 GBytes/s and an end-to-end latency for a RDMA operation of 1.87 µs. The comparison of the measurement results requires a certain carefulness, as it is debatable what is the correct way and appropriate granularity to compare the local and the remote bandwidth: should a single or several combined memory channels be cited, a single InfiniBand link or the aggregation of multiple links, and would one quote the unidirectional or bidirectional bandwidth between two machines? Ultimately, the comparison of the measurements intends to illustrate the ballpark performance gap that can be summarized as follows: as shown in Fig. 2.6, from a bandwidth perspective, local and remote memory access are in the same order of magnitude, and depending on the chosen performance metric, even on par. When it comes to comparing the latency of main memory access inside a machine and between two separate machines, there is still a gap of one to two orders of magnitude.

2.2 In-Memory Database Management Systems

Summary: This section describes a database management system where the data permanently resides in physical main memory.

As mentioned in Chap. 1, Elmasri and Navathe [EN10] describe a database system as consisting of a database and a database management system (DBMS). They define a database as "a collection of related data" and a DBMS as "a collection of programs that enables users to create and maintain a database. The DBMS is hence a general-purpose software system that facilitates the process of defining, constructing, manipulating, and sharing databases among various users and applications". According to Hellerstein and Stonebraker [HS05], IBM DB/2 [HS13], PostgreSQL [Sto87], and Sybase SQL Server [Kir96] are typical representatives of relational database management systems. The term *relational* in the context of the aforementioned DBMSs refers to the implementation of the relational data model by Codd [Cod70], which allows querying the database by using a Structured (English) Query Language initially abbreviated as SEQUEL then shortened to SQL [CB74]. This led to the SQL-standard which is regularly updated [Zem12] and is ISO-certified. Usually a SQL query is issued by an application to a DBMS. The DBMS then parses the query and creates a query plan potentially with the help of a query optimizer. The query plan is then executed by a query execution engine which orchestrates a set of database operators (such as a scan or join) in order to create the result for that query [GMUW08].

The previously mentioned DBMSs are optimized for the characteristics of disk storage mechanisms. In their paper *Main Memory Database Systems: An Overview* [GMS92] from 1992, Garcia-Molina and Salem describe a main memory database system (MMDB) as a database system where data "resides permanently in main physical memory". They argue that in a disk-based DBMS the data is cached in main memory for access, where in a main memory database system the permanently in memory residing data may have a backup copy on disk. They observe that in both cases a data item can have copies in memory and on disk at the same time. However, they also note the following main difference when it comes to accessing data that resides in main memory:

1. The access time for main memory is orders of magnitude less than for disk storage.
2. Main memory is normally volatile, while disk storage is not. However it is possible (at some cost) to construct nonvolatile memory.
3. Disks have a high, fixed cost per access that does not depend on the amount of data that is retrieved during the access. For this reason, disks are block-oriented storage devices. Main memory is not block oriented.
4. The layout of data on a disk is much more critical than the layout of data in main memory, since sequential access to a disk is faster than random access. Sequential access is not as important in main memories.
5. Main memory is normally directly accessible by the processor(s), while disks are not. This may make data in main memory more vulnerable than disk resident data to software errors. [GMS92]

In the remainder of their paper, Garcia-Molina and Salem describe the implications of memory resident data on database system design aspects such as access methods (e.g. index structures do not need to hold a copy of the indexed data, but just a pointer), query processing (e.g. focus should be on processing costs rather than minimize disk access) or recovery (e.g. use of checkpointing to and recovery from disk). IBM's Office-By-Example [AHK85], IMS/VS Fast Path [GK85] or System M from Princeton [SGM90] are presented as state-of-the-art main memory database systems at that time. The further development of memory technology in the subsequent 20 years after this statement—as illustrated in detail in Sect. 2.1.1 and Fig. 2.3—led to increased interest in main memory databases. Plattner describes in 2011 [Pla11a,Pla11b] an in-memory database system called SanssouciDB which is tailored for business applications. SanssouciDB takes hardware developments such as multi-core processors and the resulting importance of the memory wall—as explained in Sect. 2.1.2—into consideration and leverages them by allowing inter- and intra-query parallelism and exploiting cache hierarchies: an important enabler for this is the use of a columnar data layout which will be discussed in detail in the next two subsections.

2.2.1 Column- and Row-Oriented Data Layout

Summary: This subsection distinguishes between two physical database table layouts, namely storing all tuples of a record together (row-orientation) or storing all instances of the same attribute type from different tuples together (column-orientation).

As quoted in the previous subsection, Garcia-Molina and Salem [GMS92] stated that "sequential access is not as important in main memories" in comparison to disk-resident data. While the performance penalty for non-sequential data traversal is higher when operating on disk, it is also exists when accessing data that is in main memory. As described in Sect. 2.1 and as evaluated by Ailamaki et al. [ADHW99] or Boncz, Manegold, and Kersten [BMK99], the access latency from the processor to data in main memory is not truly random due to the memory hierarchy. Since the data travels from main memory through the caches to the processor, it is of relevance if all the data that sits on a cache line is truly being used (cache locality) by the processor and if a requested cache line is already present in one of the caches (temporal locality). In addition, a sequential traversal of data is a pattern that modern CPUs can detect and improve the traversal performance by loading the next to be accessed cache line while the previous cache line is still being processed: this mechanism is known as hardware prefetching (spatial locality) [HP11].

If these mechanisms can be exploited depends on the chosen data layout and the kind of data access thereupon. The two basic distinctions for a data layout are the n-ary storage model (NSM) and the decomposed storage model[CK85]. In NSM, all attributes of a tuple are physically stored together, whereas in DSM the instances of the same attribute type from different tuples are stored together. In database table terms, NSM is declared as a row-oriented data layout and DSM is called a column-

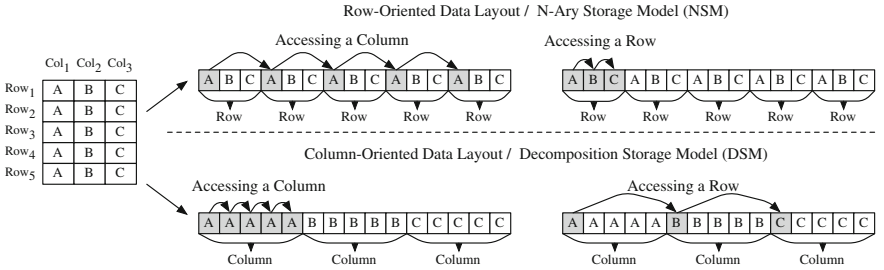

Fig. 2.7 Illustration of a row- and column-oriented data layout

oriented layout. As shown in Fig. 2.7, accessing a row or accessing a column can both leverage the benefits of locality if the data layout has been chosen accordingly. This has led to the classic distinction that databases which are intended for workloads that center around transaction processing and operate on a few rows at a time choose a row-oriented layout and read-only workloads that scan over table attributes and therefore operate on columns choose a column-oriented layout [AMH08]—the next subsection is going to discuss the classification of workloads in detail.

Another aspect in the discussion of row- and column-oriented data layout is that light weight data compression mechanisms work particularly well in a columnar data layout [AMF06]. The intention for using compression mechanisms is saving storage space and—by traversing fewer bytes for processing the same amount of information—increasing performance. The term *light-weight* describes the compression on a sequence of values (e.g. in contrast to heavy weight which refers to the compression of an array of bytes) with techniques such as dictionary compression, run-length encoding (RLE), and bit-vector encoding: these mechanisms allow the processing of a query on the compressed values and it is desirable to decompress the values as late as possible—for example, before returning the result of a query to the user. Dictionary compression is appealing in a columnar data layout as the values inside a single column have the same data type and similar semantics (in contrast to the data inside a single row). The resulting low entropy inside a column can be exploited by dictionary compression in order to reduce the required storage space drastically. Additionally, bit-vector encoding can be used to further reduce the needed storage space: e.g. if a column has up to 4096 different values in total, it can be encoded with a dictionary key size of 12 bit, and the first 60 bits of an 8-byte integer can hold five different column values. Run-length encoding can be applied to columns if they contain sequences in which the same data value occurs multiple times: instead of storing each occurrence separately, RLE allows storing the value once accompanied by the number of subsequent occurrences.

2.2.2 Transactional Versus Analytical Versus Mixed Workload Processing

Summary: This subsection explains different database workloads and how they are being affected by the choice of the data layout.

As implied in the previous subsection, database textbooks contain a distinction between online transaction processing (OLTP) and online analytical processing (OLAP) [EN10]. The term *online* expresses the instant processing of a query and delivering its result. The term *transaction* in OLTP refers to a database transaction which, in turn, has been named after the concept of a business or commercial transaction. Typical examples for OLTP applications are bank teller processing, airline reservation processing or insurance claims processing [BN97]. OLTP workloads are characterized by operating on a few rows per query with an equal mix of read and write operations. The term *analytical* in OLAP describes the intent to perform analysis on data. These analyses consist of ad-hoc queries which for example are issued to support a decision. Systems that are designed for handling OLAP workloads are also referred to as decision support systems (DSS) [Tur90]. An OLAP workload is characterized by executing aggregations over the values of a database table issued by read-mostly queries [Tho02]. OLTP and OLAP work on semantically the same data (e.g. sales transactions are recorded by an OLTP system and then analyzed with an OLAP system), yet they are typically separate systems. Initially, analytical queries were executed against the transactional system, but at the beginning of the 1990s large corporations were no longer able to do so as the performance of the transactional system was not good enough for handling both workloads at the same time. This led to the introduction of OLAP and the separation of the two systems [CCS93]. However, the resulting duplication and separation of data introduced a series of drawbacks. First, the duplication and transformation of data from the transactional to the analytical system (known as extract, transform, and load (ETL) [KC04]) requires additional processing for data denormalization and rearranging. Second, since the execution of the ETL procedure happens only periodically (e.g. once a day), the analytical processing happens on slightly outdated data. Third, the provisioning of a dedicated system for analytical processing requires additional resources.

As a consequence, there are two motivational factors for a reunification of OLTP and OLAP systems as proposed by Plattner [Pla09]: First, the elimination of the previously explained drawbacks resulting from the separation and second, the support of applications which cannot be clearly assigned to one of the workload categories, but expose a mix of many analytical queries and some transactional queries. Tinnefeld et al. [Tin09, KTGP10] elaborate that especially business applications, such as the available-to-promise (ATP) application [TMK+11], expose a mixed workload. An ATP application evaluates if a requested quantity of a product can be delivered to a customer at a requested date. This is done by aggregating and evaluating stock levels and to-be-produced goods with analytical queries, followed by transactional queries upon the reservation of products by the customer.

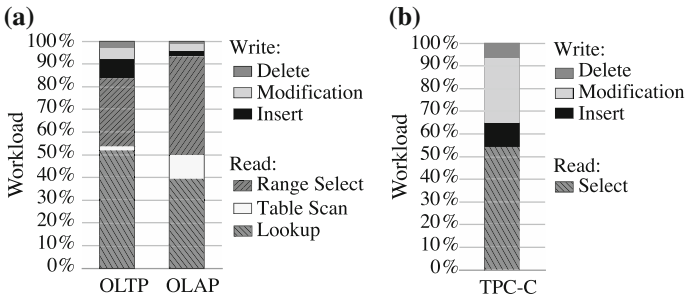

Fig. 2.8 Comparison of query distribution in analyzed business customer systems and a database benchmark (comparison by and figure taken from Krüger et al. [KKG+11]). The comparison shows that OLTP workloads on customer enterprise resource planning systems are also dominated by read operations in contrast to the common understanding that OLTP exposes an equal mix of read and write operations (as e.g. implied by the TPC-C benchmark [Raa93]). **a** Query distribution in analyzed business customer systems. **b** Query distribution in the TPC-C benchmark

In order to find a common database approach for OLTP and OLAP, it is logical to reevaluate if the previously mentioned workload characterizations are accurate [Pla09]. Krüger et al. [KKG+11] evaluated the query processing in twelve enterprise resource planning (ERP) systems from medium- and large-sized companies with an average of 74,000 database tables per customer system. As shown in Fig. 2.8a, the distribution of queries in the OLAP category is as expected: over 90 % of the queries are read operations dominated by range select operations. The OLTP queries, however, also consist of over 80 % read queries dominated by lookup operations. Only 17 % of the overall queries result in write operations: a contradiction to the common understanding that OLTP consists of an equal mix of read and write queries. This misconception can be visualized by looking at the query distribution of the TPC-C benchmark (which is the standard benchmark for OLTP systems [Raa93]) as shown in Fig. 2.8b: almost 50 % of the queries are write operations, while range select and table scan operations are not included at all.

The drawbacks of the separation of OLTP and OLAP, the missing support for applications that expose a mixed workload, and the misconception of the nature of OLTP in the first place drove Plattner to design a common database approach for OLTP and OLAP [Pla09, Pla11b, Pla11a]: SanssouciDB provides a common database for OLTP and OLAP and provides adequate performance by leveraging modern hardware technology: mainly the storage of all data in main memory and the utilization of multi-core processors. Although it supports a column- and row-oriented data layout, it puts heavy emphasis on the use of a columnar layout as (i) it is the best match for OLAP workloads, (ii) it has been shown that even OLTP applications have a portion of over 80 % read queries, and (iii) also mixed workloads are dominated by analytical queries.

2.2.3 State-of-the-Art In-Memory Database Management Systems

Summary: This subsection lists state-of-the-art in-memory database management systems.

In this subsection we present a selection of academic and industry in-memory database management systems which are considered state-of-the-art in alphabetical order. We describe their main characteristics, starting with the systems from academia:

- **HyPer (Technical University Munich)** [KNI+11, Neu11] is a main memory database hybrid system that supports row and columnar data layout with the goal of supporting OLTP and OLAP workloads. To guarantee good performance for both workloads simultaneously, HyPer creates snapshots of the transactional data with the help of hardware-assisted replication mechanisms. Data durability is ensured via logging onto a non-volatile storage, high-availability can be achieved by deploying a second hot-standby server. The separate data snapshots for OLTP and OLAP workloads allow conflict-free multi-threaded query processing as well as the deployment to several servers to increase the OLAP throughput [MRR+13].
- **HYRISE (Hasso-Plattner-Institut)** [GKP+10, GCMK+12] is a main memory storage engine that provides dynamic vertical partitioning of the tables it stores. This means that fragments of a single table can either be stored in a row- or column-oriented manner with the intention of supporting OLTP, OLAP, and mixed workloads. HYRISE features a layout algorithm based on a main memory cost model in order to find the best hybrid data layout for a given workload.
- **MonetDB (Centrum Wiskunde and Informatica)** [BGvK+06, BKM08a] is a main memory columnar database management system that is optimized for the bandwidth bottleneck between CPU and main memory. The optimizations include cache-conscious algorithms, data compression, and the modeling of main memory access costs as an input parameter for query optimization. MonetDB is purely intended for executing OLAP workloads and does not support transactions or durability.

The following main memory database systems from industry are presented:

- **IBM solidDB** [MWV+13] is a relational database management system that can either be deployed as an in-memory cache for traditional database systems, such as IBM's DB2 [HS13], or as a stand-alone database. In both cases, it exposes an SQL interface to applications. When deployed as a stand-alone database it offers an in-memory as well as a disk-based storage engine. The in-memory engine uses a trie data structure for indexing, where the nodes in the trie are optimized for modern processor cache sizes. The trie nodes point to data stored consecutively in arrays in main memory. When using the in-memory storage, snapshot-consistent checkpointing [WH 1] to disk is used for ensuring durability. IBM is positioning solidDB as a database for application areas such as banking, retail, or telecom [MWV+13].

- **Microsoft SQL Server** has two components which are tailored for in-memory data management: the in-memory database engine Hekaton [DFI+13], which is optimized for OLTP workloads, and xVelocity [Doc13], which is a columnstore index and an analytics engine designed for OLAP workloads. Hekaton stores data either via lock-free hash tables [Mic02] or via lock-free B-trees [LSS13]. In order to improve transactional throughput, Hekaton is able to compile requests to native code. xVelocity offers an in-memory columnar index—not a memory resident columnar storage—that supports data compression and can be utilized by the xVelocity analytics engine, to provide analytical capabilities in conjunction with Microsoft Excel. However, Microsoft SQL Server also offers an updateable columnar storage engine [LCF+13] which stores its data on SSD/disk. Microsoft SQL server is positioned as a general-purpose database.
- **SAP HANA** [FCP+12] is an in-memory database that supports row- and column-oriented storage in a hybrid engine for supporting OLTP, OLAP as well as mixed workloads (see Sect. 2.2.2). In addition, it features a graph and text processing engine for semi- and unstructured data management within the same system. HANA is mainly intended to be used in the context of business applications and can be queried via SQL, SQL Script, MDX, and other domain-specific languages. It supports multiversion concurrency control and ensures durability by logging to SSD. A unique aspect of HANA is its support of transactional workloads via the column store [KGT+10]: the highly compressed column store is accompanied by an additional write-optimized buffer called delta store. The content of the delta is periodically merged into the column store. This architecture provides both fast read and write performance.
- **VoltDB** [SW13] is an in-memory database designed for OLTP workloads and implements the design of the academic H-Store project [KKN+08]. VoltDB persists its data in a row-oriented data layout in main memory and applies checkpointing and transaction logging for providing durability. It boosts transactional throughput by analyzing transactions at compile time, and compiles them as stored procedures which are invoked by the user at run-time with individual parameters. It is designed for a multi-node setup where data is partitioned horizontally and replicated across nodes to provide high availability. VoltDB is relatively young (first release in 2010) and positions itself as a scalable database for transaction processing.

2.3 Parallel Database Management Systems

Summary: This section introduces parallel database management systems which are a variation of distributed database management systems with the intention to execute a query as fast as possible.

As mentioned in Chap. 1, Özsu and Valduriez [ÖV11] define a *distributed database* as

> a collection of multiple, logically interrelated databases distributed over a computer network. A distributed database management system is then defined as the software system that permits the management of the distributed database system and makes the distribution transparent to the users.

The two important terms in these definitions are *logically interrelated* and *distributed over a computer network*. Özsu and Valduriez mention that the typical challenges tackled by distributed database systems include, for example, the aspect of data placement, distributed query processing, distributed concurrency control, and deadlock management, ensuring reliability and availability as well as the integration of heterogeneous databases.

The term *distributed over a computer network* makes no assumption whether the network is a wide area or local area network. A database system that is running on a set of nodes which are connected via a fast network inside a cabinet or inside a data center can be classified as a *parallel database system*. According to Özsu and Valduriez [ÖV11] this can be seen as an revision and extension of a distributed database system. According to DeWitt and Gray [DG92], a parallel database system exploits the parallel nature of an underlying computing system in order to provide high-performance and high-availability.

2.3.1 Shared-Memory Versus Shared-Disk Versus Shared-Nothing

Summary: This subsection introduces and compares different parallel database management system architectures and reflects those in the context of main memory databases.

As briefly summarized in Chap. 1, one fundamental and much debated aspect of a parallel DBMS is its architecture. The architecture influences how the available hardware resources are shared and interconnected. As shown in Fig. 2.9, there

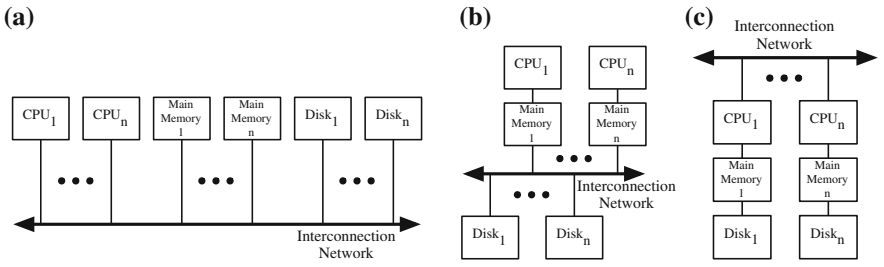

Fig. 2.9 Three different parallel DBMS architectures. **a** Shared-Memory. **b** Shared-Disk. **c** Shared-Nothing

are three different parallel DBMS textbook architectures [ÖV11, DG92, DMS13]: shared-memory, shared-storage (or shared-disk or shared-data), and shared-nothing:

Shared-memory (or shared-everything) (see Fig. 2.9a) is an architectural approach where all processors share direct access to any main memory module and to all disks over an interconnection. Examples for shared-memory DBMS are DBS3 [BCV91] and Volcano [Gra94b]. Shared-memory provides the advantages of an architecturally simple solution: there is no need for complex distributed locking or commit protocols as the lock manager and buffer pool are both stored in the memory system where they can be accessed uniformly by all processors [ERAEB05, DMS13]. In addition, the shared-memory approach is great for parallel processing: inter-query parallelism is an inherent property as all processors share all underlying resources, which means that any query can be executed by any processor. Intra-query parallelism can also be easily achieved due to the shared resources. Shared-memory has two major disadvantages: the extensibility is limited as all the communication between all resources goes over a shared interconnection. For example, a higher number of processors causes conflicts with the shared-memory resource which degrades performance [TS90]. The biggest drawback of shared-memory is its limited availability. Since the memory space is shared by all processors, a memory fault will affect many processors and potentially lead to a corrupted or unavailable database. Although the research community addressed this problem by work on fault-tolerant shared-memory [SMHW02], the shared-memory architecture never had much traction outside academic work and had its peak in the nineties (in terms of available products in industry and published research papers).

Shared-storage (or shared-disk or shared-data) (see Fig. 2.9b) is an architectural approach where processors each have their own memory, but they share access to a single collection of disks. The term shared-disk is a bit confusing in the way that it suggests that rotating disks are an integral part. This is not the case, but hard drive disks were the commonly used storage device when the term was coined. Nowadays a shared-storage architecture can be realized by storing data on disk, SSD, or even keeping it main memory resident (e.g. see Texas Memory Systems RamSan-440 [ME13] or RAMCloud [OAE+11]), typically in the form of a storage area network (SAN) or a network-attached storage (NAS). However, each processor in a shared-storage approach can copy data from the database in its local memory for query processing. Conflicting access can be avoided by global locking or protocols for maintaining data coherence [MN92]. Examples for shared-storage systems are IBM IMS [KLB+12] or Sybase IQ [Moo11]. Shared-storage brings the advantage that it is very extensible as an increase in the overall processing and storage capacity can be done by adding more processors respectively disks. Since each processor has its own memory, interference on the disks can be minimized. The coupling of main memory and processor results in an isolation of a memory module from other processors which results in better availability. As each processor can access all data, load-balancing is trivial (e.g. distributing load in a round-robin manner over all processors). The downsides are an increased coordination effort between the processors in terms of distributed database system protocols, and the shared-disks becoming a bottleneck

(similar to the shared-memory approach, sharing a resource over an interconnection is always a potential bottleneck).

Shared-nothing (see Fig. 2.9c) is an architectural approach where each memory and disk is owned by some processor which acts as a server for that data. The Gamma Database Machine Project [DGS+90] or VoltDB [SW13] are examples for shared-nothing architectures. The biggest advantage of a shared-nothing architecture is reducing interferences and resource conflicts by minimizing resource sharing. Operating on the data inside a local machine allows operating with full raw memory and disk performance, high-availability can be achieved by replicating data onto multiple nodes. With careful partitioning across the different machines, a linear speed-up and scale-up can be achieved for simple workloads [ÖV11]. A shared-nothing architecture has also its downsides: the complete decoupling of all resources introduces a higher complexity when implementing distributed database functions (e.g. providing high-availability). Load balancing also becomes more difficult as it is highly dependent on the chosen partition criteria, which makes load balancing based on data location and not actual load of the system. This also impacts the extensibility as adding new machines requires a reevaluation and potentially a reorganization of the existing data partitioning.

When comparing the different architectures one can conclude that there is no better or worse and no clear winner. The different architectures simply offer different trade-offs with various degrees of trading resource sharing against system complexity. Consequently, this is a much debated topic. For example, Rahm [Rah93] says that

> A comparison between shared-disk and shared-nothing reveals many potential benefits for shared-disk with respect to parallel query processing. In particular, shared-disk supports more flexible control over the communication overhead for intratransaction parallelism, and a higher potential for dynamic load balancing and efficient processing of mixed OLTP/query workloads.

DeWitt, Madden, and Stonebraker [DMS13] argue that

> Shared-nothing does not typically have nearly as severe bus or resource contention as shared-memory or shared-disk machines, shared-nothing can be made to scale to hundreds or even thousands of machines. Because of this, it is generally regarded as the best-scaling architecture. Shared-nothing clusters also can be constructed using very low-cost commodity PCs and networking hardware.

Hogan [Hog13] summarizes his take on the discussion with

> The comparison between shared-disk and shared-nothing is analogous to comparing automotive transmissions. Under certain conditions and, in the hands of an expert, the manual transmission provides a modest performance improvement ... Similarly, shared-nothing can be tuned to provide superior performance ... Shared-disk, much like an automatic transmission, is easier to set-up and it adjusts over time to accommodate changing usage patterns.

When reviewing this discussion in the context of parallel main memory databases, there is a clearer picture: most popular systems such as MonetDB, SAP HANA or VoltDB use a shared-nothing architecture and not a shared-storage approach. In the past, there was a big performance gap between accessing main memory inside a

machine and in a remote machine. Consequently, a performance advantage that has been achieved by keeping all data in main memory should not vanish by sending much data over a substantially slower network interconnect. As shown in Sect. 2.1.3, the performance gap between local and remote main memory access performance is closing, which paves the way for discussing a shared-storage architecture for a main memory database.

2.3.2 State-of-the-Art Parallel Database Management Systems

Summary: This subsection presents state-of-the-art shared-storage and shared-nothing parallel database management systems.

In this subsection we present a selection of disk-based and main memory-based as well as shared-storage and shared-nothing parallel database management systems and the different variations thereof. Some of the systems were previously introduced, but this subsection focuses on their ability to be deployed on several servers. We start with shared-storage parallel database management systems:

- **IBM DB2 pureScale** [IBM13] is a disk-based shared-storage solution for IBM's row-oriented DB2 [HS13]. It allows the creation of a parallel DBMS consisting of up to 128 nodes where each node is an instance of DB2 and all nodes share access to a storage system that is based on IBM's General Parallel File System (GPFS) [SH02]. Each database node utilizes local storage for caching or maintaining a bufferpool. In addition to the local bufferpools per node, there is also a global bufferpool that keeps record of all pages that have been updated, inserted or deleted. This global bufferpool is used in conjunction with a global lock manager: before any node can make for example an update, it has to request the global lock. After data modification, the global lock manager invalidates all local copies of the respective page in the local memory of the nodes. PureScale offers so called cluster services which, for example, orchestrate the recovery process in the event of a downtime. GPFS stores the data in blocks of a configured size and also supports striping and mirroring to ensure high-availability and improved performance.
- **MySQL on MEMSCALE** [MSFD11a,b] is shared-storage deployment of MySQL on MEMSCALE, which is a hardware-based shared-memory system that claims to scale gracefully by not sharing cores nor caches, and therefore working without a coherency protocol. This approach uses the main memory storage engine of MySQL (known as heap or memory) and replaces the common *malloc* by a library function that allocates memory from the MEMSCALE shared-memory. As a consequence, all properties of MySQL are still present so that queries can be executed in a multithreaded, ACID compliant manner with row-level locking.
- **MySQL on RamSan** [ME13] is a shared-storage solution where MySQL utilizes the storage space provided by a storage area network that keeps all data resident in main memory called RamSan. RamSan was originally developed by Texas Memory Systems (now acquired by IBM). It acts as a traditional storage area network, but depending on the configuration keeps all data in main memory

modules such as DDR, and uses additional SSDs for non-volatile storage. RamSan provides the advantage that it is transparent to a database system, such as MySQL, that main memory-based storage is being used, but incorporates the downside that data access is not optimized for main memory.

- **ScaleDB** [Sca13] implements a shared-storage architecture that consists of three different entities, namely database nodes, storage nodes, and a cluster manager. Each database node runs an instance of MySQL server that is equipped with a custom ScaleDB storage engine by utilizing the MySQL feature of pluggable storage engines [MyS13a]. That results in a multithreaded, disk-based, ACID compliant engine that supports row-level locking and operates in read committed mode. All database nodes share common access to the data inside the storage nodes. ScaledDB manages the data in block-structured files with each individual file being broken into blocks of a fixed size. These blocks are stored twice for providing high-availability and are distributed randomly across the storage nodes. In addition, ScaleDB also features a cluster manager that is a distributed lock manager that synchronizes lock requests of different nodes.
- **Sybase IQ** [Syb13a] is a shared-storage, columnar, relational database system that is mainly used for data analytics. As depicted in the Sybase IQ 15 sizing guide [Syb13b], a set of database nodes accesses a commonly shared storage that holds all data. Among the database nodes is a primary server (or coordinator node) that manages all global read-write transactions and maintains the global catalog and metadata. In order to maximize the throughput when operating on the shared storage system, Sybase IQ strips data with the intention of utilizing as many disks in parallel as possible.

The following shared-nothing parallel database management systems are presented:

- **C-Store** [SAB+05] and its commercial counterpart **Vertica** [LFV+12] are disk-based columnar parallel DBMSs based on a shared-nothing architecture. The data distribution in Vertica is done by splitting tuples among nodes by a hash-based ring style segmentation scheme. Within each node, tuples are physically grouped into local segments which are used as a unit of transfer when nodes are being added to or removed from the cluster in order to speed up the data transfer. Vertica allows defining how often a data item is being replicated across the cluster (called k-saftey), realizing thereby high-availability, and allows the operator of the cluster to set the desired tradeoff between hardware costs and availability guarantees.
- **MySQL Cluster** [DF06] enables MySQL (MySQL [MyS13b] is one of the most popular open-source relational DBMS) to be used as a shared-nothing parallel DBMS. MySQL Cluster partitions data across all nodes in the system by hash-based partitioning according to the primary key of a table. The database administrator can choose a different partitioning schema by specifying another key of a table as partitioning criteria. In addition, data is synchronously replicated to multiple nodes for guaranteeing availability. Durability is ensured in a way that each node writes logs to disk in addition to checkpointing the data regularly. In a MySQL cluster, there are three different type of nodes: a management node that is

used for configuration and monitoring of the cluster, a data node which stores parts of the tables, and a SQL node that accepts queries from clients and automatically communicates with all other nodes that hold a piece of the data needed to execute a query.

- **Postgres eXtensible Cluster** (Postgres-XC) [Pos13, BS13] is a solution to deploy PostgreSQL as a disk-based, row oriented, shared-nothing parallel DBMS. A Postgres-XC cluster consists of three different types of entities: a global transaction manager, a coordinator, and datanodes. The global transaction manager is a central instance in a Postgres-XC cluster and enables multi-version concurrency control (e.g. by issuing unique transaction IDs). The coordinator accepts queries from an application and coordinates their execution by requesting a transaction ID from the global transaction manager, determining which datanodes are needed for answering a query and sending them the respective part of the query. The overall data is partitioned across datanodes where each datanode executes (partial) queries on its own data. Data can also be replicated across the datanodes in order to provide high-availability.
- **SAP HANA** [FML+12] is an in-memory parallel DBMS based on a shared-nothing architecture [LKF+13]. A database table in HANA can be split by applying round-robin, hash- or range-based partitioning strategies: the database administrator can assign the resulting individual partitions to individual HANA nodes either directly or based on the suggestions of automated partitioning tools. There are two different types of nodes: a coordinator node and a worker node. A database query issued by a client gets send to a coordinator node first. A coordinator is responsible for compiling a distributed query plan based on data locality or issuing global transaction tokens. The query plan then gets executed by a set of worker nodes where each worker node operates on its local data. HANA also features a distributed cost-based query optimizer that lays out the execution of single database operators which span multiple worker nodes. High-availability is ensured by hardware redundancy which allows to provide a stand-by server for a worker node that takes over immediately if the associated node fails [SAP13]. Durability is ensured by persisting transaction logs, savepoints, and snapshots to SSD or disk in order to recover from host failures or to support the restart of the complete system.
- **MonetDB** [BGvK+06] is an in-memory columnar DBMS that can be deployed as a shared-nothing cluster. MonetDB is a research project, it comes with the necessary primitives such as networking support and setting up a cluster by connecting individual nodes each running MonetDB. This foundation can be used to add shared-nothing data partitioning features and distributed query optimizations for running data analytics [MC13]. MonetDB is also used as a platform for researching novel distributed data processing schemes: for example the Data Cyclotron [GK10] project creates a virtual network ring based on RDMA-enabled network facilities where data is perpetually being passed through the ring, and individual nodes pick up data fragments for query processing.
- **Teradata Warehouse** [Ter13, CC11] is a shared-nothing parallel database system used for data analytics. The architecture consists of three major component types: a parsing engine, an access module processor (AMP), and the BYNET

framework. The parsing engine accepts queries from the user, creates query plans and distributes them over the network (BYNET framework) to the corresponding access module processors. An AMP is a separate physical machine that runs an instance of the Teradata Warehouse database management system which solely operates on the disks of that machine. The disks inside an AMP are organized in redundant arrays to prevent data loss. The data is partitioned across all AMPs by a hash partitioning schema in order to distribute the data evenly and reduce the risk of bottlenecks. Teradata warehouse allows configuring dedicated AMPs for hot standby which can seamlessly take over in the case of a failure of an active AMP.

- **VoltDB** [SW13] is a row-oriented in-memory database that can be deployed as a shared-nothing parallel database system. In such a cluster, each server runs an instance of the VoltDB database management system. Tables are partitioned across nodes by hashing the primary key values. In addition, tables can also be replicated across nodes for performance and high-availability. For ensuring high-availability, three mechanisms are in place: k-safety which allows to specify the number k of data replicas in the cluster. Network fault detection evaluates in the case of a network fault which side of the cluster should continue operation based on the completeness of the data. Live node rejoin allows nodes when they restart after a failure to be reintroduced to the running cluster and retrieve their copy of the data. Durability is ensured via snapshots to disks in intervals and command logging.

2.3.3 Database-Aware Storage Systems

Summary: This subsection introduces database-aware storage systems which are storage systems that support the execution of database operations in order to reduce network communication.

The previous discussion of shared-storage versus shared-nothing architectures describes that each architecture has its advantages: one advantage of a shared-nothing architecture is that a processor performs data operations on the data that is inside the same machine without the need for transferring the data over a network first. Database-aware storage systems [RGF98, Kee99, SBADAD05] aim at bringing that advantage to shared-storage systems by executing database operators directly in the storage system. This approach is based on the idea of active storage/active disks/intelligent disks [AUS98, KPH98] where the computational power inside the storage device is being used for moving computation closer to the data.

The American National Standards Institute (ANSI) Object-based Storage Device (OSD) T10 standard describes a command set for the Small Computer System Interface (SCSI) [sta04] that allows communication between the application and the storage system. This, in turn, is the foundation for the work of Raghuveer, Schlosser, and Iren [RSI07] who use this OSD interface for improving data access for a database application by making the storage device aware of relations, in contrast to just returning blocks of data. However, they do not support the execution of application/database code inside the storage device. This is done in the Diamond project

[HSW+04], as it applies the concept of *early discard* in an active storage system. *Early discard* describes the rejection of to be filtered out data as early as possible. Diamond supports the processing of such filters inside an active disk and makes sure that not requested data is discarded before it reaches the application, such as a database system. Riedel, Gibson, and, Faloutsos [RGF98] evaluate the usage of active disks for a different set of application scenarios including nearest neighbor search in a database and data mining of frequent sets. They conclude that the processing power of a disk drive will always be inferior to server CPUs, but that a storage system usually consists of a lot of disks resulting in the advantage of parallelism, and that combining their computational power with the processors inside the server results in a higher total processing capacity. In addition, the benefit of filtering in the disk reduces the load on the interconnect and brought significant performance advantages for their application scenarios.

An example for a current system that exploits the possibilities of database-aware storage systems is Oracle's Exadata which combines a database and a storage system inside a single appliance: the Exadata storage system supports *Smart-Scan* and Offloading [SBADAD05]. *Smart-Scan* describes the possibility to execute column projections, predicate filtering, and index creation inside the storage system. Offloading enables the execution of more advanced database functions inside the storage system, such as simple joins or the execution of mathematical or analytical functions.

2.3.4 Operator Placement for Distributed Query Processing

Summary: This subsection describes and discusses query, hybrid, and data shipping which are three different approaches how the resources from client and server can be utilized for processing a query in a distributed database management system.

When using a database-aware storage system, the resources from the storage system, as well as the resources from the DBMS, can be utilized for query processing. In the field of distributed query processing, this problem is classified as the exploitation of client resources in the context of client-server database systems. As this problem is originated in a setting where the client is an application and the DBMS acts as a server, the remainder of this section presents a detailed description of the problem including the related work and discusses it in conjunction with database-aware storage systems, where the DBMS is the client and the server is a storage system.

Tanenbaum explains the client-server model by stating that "a client process sends a request to a server process which then does the work and sends back the answer" [Tan07]. Based on that definition, Kossmann gives in his seminal paper *The State of the Art in Distributed Query Processing* [Kos00] (which is in form and content the blueprint for the remainder of this subsection) a description of the client resource exploitation problem:

> The essence of client-server computing is that the database is persistently stored by server machines and that queries are initiated at client machines. The question is whether to execute a query at the client machine at which the query was initiated or at the server machines that

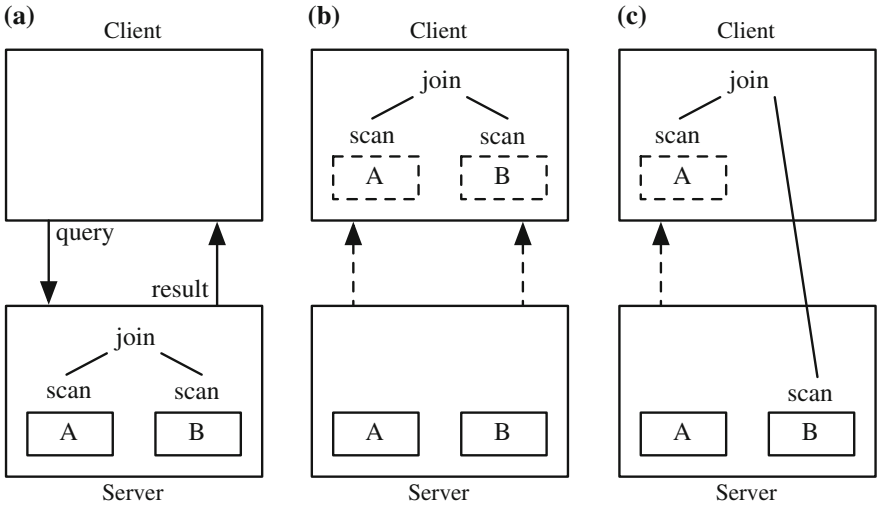

Fig. 2.10 Illustration of query, data, and hybrid shipping (Figure taken from [Kos00]). **a** Query Shipping. **b** Data Shipping. **c** Hybrid Shipping

store the relevant data. In other words, the question is whether to move the query to the data (execution at servers) or to move the data to the query (execution at clients).

This results in the following three approaches:

Query shipping (see Fig. 2.10a) is an approach where the client sends the query in its entirety to the server, the server processes the query and sends back the result. This is the approach that is typical for example for relational DBMSs such as Microsoft SQL Server or IBM DB2.

Data shipping (or data pull) (see Fig. 2.10b) is the opposite solution where the client consumes all the needed data from the server and then processes the query locally. This results in an execution of the query where it originated. Object-oriented databases often work after the data shipping principle as the client consumes the objects as a whole and then does the processing.

Hybrid shipping (see Fig. 2.10c) is a combination of the two previous approaches. As shown by Franklin, Jónsson, and Kossmann [FJK96], hybrid shipping allows executing some query operators at the server side, but also pulling data to and processing it at the client side. As shown in Fig. 2.10c, data region A is being pulled and scanned at the client's side. Data region B is scanned at the server's side, and the results are then transferred to the client where they are joined with the results from the scan on data region A.

The three different approaches imply a number of design decisions when creating a model that supports the decision where the execution of each individual database operator that belongs to a query is being placed. Those decisions are: (a) the site selection itself which determines where each individual operator is being executed, (b) where to decide the site selection, (c) what parameters should be considered when doing the site selection, and (d) when to determine the site selection.

Table 2.2 Site selection for different classes of database operators for query, data, and hybrid shipping (Figure taken from [Kos00])

Database operator	Query shipping	Data shipping	Hybrid shipping
Display	Client	Client	Client
Update	Server	Client	Client or server
Binary operators (e.g. join)	Producer of left or right input	Consumer (i.e. client)	Consumer or producer of left or right input
Unary operators (e.g. sort, group-by)	Producer	Consumer (i.e. client)	Consumer or producer
Scan	server	Client	Client or server

Site selection in conjunction with the client resource exploitation problem says that each individual database operator has a site annotation that indicates where this operator is going to be executed [Kos00]. As shown in Table 2.2, different annotations are possible per operator class: display operations return the result of a query which always has to happen at the client. All other operators in a data shipping approach are executed at the client site as well or in other words at the site where the data is consumed. Query shipping executes all operators at the server site or where the data is produced (e.g. by the execution of a previous operator). Hybrid shipping supports both annotations. The question **where** to make the site selection depends on several factors, most notably the number of servers in the system: if there is only one server, it makes sense to let the server decide the site selection as it knows its own current load [HF86]. If there are many servers, there might be no or only little information gained by executing it at the server site as a single server has no complete knowledge of the system. In addition, the site selection itself is also an operation that consumes resources which can be scaled with the number of clients if executed at the client. **What** information is considered for the site selection depends on the nature of the site selection algorithm. For example, for a cost-based approach one might want to consider information about the database operation itself, such as the amount of data to be traversed or the selectivity, the hardware properties of the client, the server, as well as the interconnect or information about the current load of the client and/or the server. If the site selection algorithm is a heuristic, simple information such as the class of the current database operation and the selectivity might be sufficient (e.g. executing a scan operator at the server site if the selectivity is greater than x). Three different approaches are possible with regards to deciding **when** site selection occurs: a static, a dynamic, and a two-step optimization approach. A static approach can be chosen if the queries are known and compiled at the same time when the application itself is being compiled. This allows also to decide on the site selection at compile time. Only in exceptional situations, such as a change in the predetermined queries, doesa reevaluation of the site selection occur [CAK+81]. This approach works well if queries and workload are static, but performs poorly with a fluctuating or changing workload. A simple dynamic approach generates alternative site selection plans at design time and chooses the site selection plan during query

execution that for example best matches the assumptions about the current load of the system. If that repeatedly results in poor performance, a new set of site selection plans can be generated [CG94]. This can be especially useful if certain servers are not responsive [UFA98] or if the initial assumptions of e.g. the sizes of intermediate results turned out to be wrong [KD98]. Two-step optimization is a more advanced dynamic approach that determines the site selection just before the query execution. This is achieved by a decomposition of the overall query execution into two steps [Kos00]. First, a query plan is generated that specifies the join order, join methods, and access paths. This first step has the same complexity as the query plan creation in a centralized system. Second, right before executing the query plan, it is determined where every operator is being executed. As the second step only carries out the site selection for a single query at a time, its complexity is reasonably low and can also be done during query execution. As a consequence, it is assumed that this approach reduces the overall complexity of distributed query optimization and is hereby popular for distributed and parallel database systems [HM95, GGS96]. The advantages of the two-step optimization are the aforementioned low complexity and the ability to consider the current state of the system at the time of query execution which can be used for workload distribution [CL86] and to exploit caching [FJK96]. The downside of decoupling query plan creation and site selection is ignoring the placement of data across servers during query plan creation as it might result in a site selection plan with unnecessary high communication costs [Kos00].

The previous explanations are a general take on the client resource exploitation and the resulting site selection problem. They include the underlying assumption that every operator in a query can be executed at the client or the server. However, this might not be the case as not all database operators are available at both sites. This limits the site selection scope to a subset or a set of classes of database operators depending on which are available at both sites. In addition, the execution order of the operators according to the query plan also limits the site selection (e.g. a query plan foresees that two relations are scanned and the results of the scan operations are joined). The scan operators are available at the client and server sites, the join operator is only available at the server. In this case, executing the scans at the client site results in an unreasonably high communication overhead as both relations have to be shipped in their entirety to the client and then the scan results have to be shipped back to the server for processing the join. The availability of operators at a site depends on which kind of system acts as the client and the server. If the client is an application and the server is a relational database management system, then the entirety of database operators is available at the server site. The default mode is to ship the query to the DBMS and retrieve the result. It is possible that some operators are also available at the client site (e.g. a scan and a group-by operator). That allows hybrid shipping for queries that facilitate both, and makes data shipping an option for queries that only consists of these two types of operators. In such a setting, all possible queries can be executed with query shipping, a subset of them with hybrid shipping and an even smaller subset with data shipping (as depicted in Fig. 2.11). The situation is reversed when the client is a relational DBMS and the server is a database-aware storage system. If the database-aware storage system has a scan

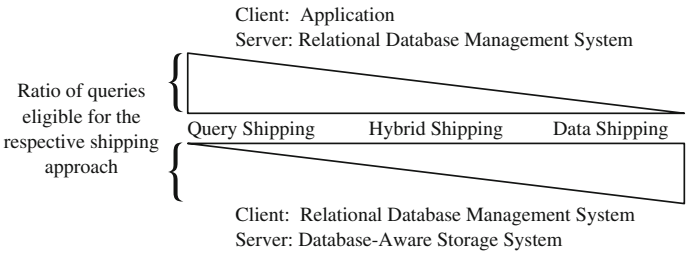

Fig. 2.11 Depicting the ratio of queries which are eligible for query, hybrid or data shipping in correspondence with a relational DBMS either acting as server or client

operator implemented, all queries can be executed with data shipping, queries which involve a scan can be executed via hybrid shipping and queries solely consisting of scans have the option of using a query shipping approach.

2.4 Cloud Storage Systems

Summary: This section introduces and classifies cloud storage systems and describes how they differ from traditional database and file systems.

The paradigm of cloud computing describes the provisioning of information technology infrastructure, services, and applications over the Internet. Typical characteristics are the on-demand availability of such resources, their quick adaption to changing workloads, and the billing based on actual usage. From a data management perspective, two different types of cloud storage systems have been created to manage and persist large amounts of data created and consumed by cloud computing applications: (a) So called NoSQL[2] systems include distributed key-value stores, such as Amazon Dynamo [DHJ+07] or Project Voldemort, wide column stores such as Google Bigtable [CDG+06] or Cassandra [LM10] as well as document and graph stores. (b) Distributed file systems such as Google File System (GFS) [GGL03] or Hadoop Distributed File System (HDFS) [SKRC10]. The remainder of this section explains the motivation for building these cloud storage systems, and their underlying concepts, as well as how they differ from traditional database and file systems.

The previous section explains that parallel database management systems combine the resources of multiple computers to accelerate the execution of queries as well as increase the overall data processing capacity. Such systems were greatly challenged with the advent of e-commerce and Internet-based services offered by companies

[2]There is no explicit explanation what the abbreviation NoSQL stands for, but it is most commonly agreed that it means "not only SQL". This term does not reject the query language SQL, but rather expresses that the design of relational database management systems is unsuitable for large-scale cloud applications [Bur10] (see Eric Brewer's CAP theorem [Bre00, Bre12] as explained later in this Section).

such as Amazon, eBay or Google in the mid-nineties. These companies not only grew rapidly in their overall size, they also did it in a comparatively short period of time with unpredictable growth bursts. This development put big emphasis on the aspects of *scalability* and *elasticity* in the context of data storage and processing systems. Scalability is defined as

> a desirable property of a system, which indicates its ability to either handle growing amounts of work in a graceful manner or its ability to improve throughput when additional resources (typically hardware) are added. A system whose performance improves after adding hardware, proportionally to the capacity added, is said to be a scalable system [AEADE11].

Elasticity hardens the scalability property as it focuses on the quality of the workload adaption process—e.g. when resources have been added—in terms such as money or time. Elasticity can be defined as

> the ability to deal with load variations by adding more resources during high load or consolidating the tenants to fewer nodes when the load decreases, all in a live system without service disruption, is therefore critical for these systems. ... Elasticity is critical to minimize operating costs while ensuring good performance during high loads. It allows consolidation of the system to consume less resources and thus minimize the operating cost during periods of low load while allowing it to dynamically scale up its size as the load decreases [AEADE11].[3]

For example, the Internet-based retailer Amazon initially used off-the-shelf relational database management systems as the data processing backend for their Internet platform. The Amazon Chief Technology Officer Werner Vogels explains [Vog07] that Amazon had to perform short-cycled hardware upgrades to their database machines in their early days. Each upgrade would only provide sufficient data processing performance for a couple of months until the next upgrade was due to the company's extreme growth. This was followed by attempts to tune their relational database systems by simplifying the database schema, introducing various caching layers or partitioning the data differently. At some point the engineering team at Amazon decided to evaluate the data processing needs at Amazon and create their own data processing infrastructure accordingly. These data processing needs reveal [Vog07] that at Amazon e.g. about 65 % of the data access is based on the primary key only, about 15 % of the data access exposes lot of writes in combination with the need for strong consistency, and 99.9 % of the data access has a need for low latency where a response time of less than 15 ms is expected. The variety in their data processing needs is covered by a set of solutions where the most prominent one is Amazon Dynamo.

Amazon Dynamo [DHJ+07] is a distributed key-value store. Since the majority of data operations at Amazon encompass primary key access only and require low latency, Dynamo is designed for storing and retrieving key-value pairs (also referred to as objects). Dynamo implements a distributed hash table where the different nodes that hold data of that hash table are organized as a ring. The data distribution is done

[3] The definitions and usage of the terms scalability and elasticity are much discussed in the computer science community as they are not strictly quantifiable [Hil90].

via consistent hashing. However, a hash-based data partitioning onto physical nodes would lead to a non-uniform data and load distribution due to the random position assignment of each node in the ring and the basic algorithm is oblivious to the heterogeneity in the performance of nodes. This is addressed by the introduction of virtual nodes (which can be thought of as namespaces), where a single virtual node is being mapped onto different physical nodes: this allows for providing high-availability and elasticity by changing the number of physical nodes assigned to a virtual node upon workload changes. As a result, nodes can be added and removed from Dynamo without any need for manual partitioning and redistribution. Consistency is facilitated by object versioning [Lam78]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. These features in combination make Dynamo a scalable, highly-available, completely decentralized system with minimal need for manual administration. Amazon Dynamo is just one example of a distributed key-value store, but it is conceptually similar to other popular distributed key-value stores such as Riak [Klo10] or Project Voldemort [SKG+12]. Google Bigtable [CDG+06] and Cassandra [LM10] are examples for wide column stores (also known as extensible record stores [Cat11]): for example Google Bigtable implements a sparse, distributed, persistent multi-dimensional map indexed by a row key, a column key, and some kind of versioning information such as a timestamp or versioning number. This multi-dimensional map is partitioned in a cluster in the following way: the rows of a map are split across nodes with the assignment being done by ranges and based on hashing. Each column of a map is described as a column family, the contained data is of the same type and its data is being distributed over multiple nodes. Such a column family must not be mistaken for a relational database column that implies that related data is physically collocated, but can be seen more as a namespace. The motivation for using such wide column stores over simple distributed key-value stores is the more expressive data model, that provides a simple approach to model references between data items [CDG+06].

How a cloud storage system such as Amazon Dynamo differs from a relational DBMS and how it addresses the previously mentioned scalability shortcomings can be illustrated with the Consistency, Availability, and Partition Tolerance (CAP) theorem by Eric Brewer [Bre00, Bre12]. In this theorem

> consistency means that all nodes see the same data at the same time, availability is a guarantee that every request receives a response about whether it was successful or failed and partition tolerance lets the system continue to operate despite arbitrary message loss.

The theorem states that a distributed system can provide two properties at the same time, but not three. Relational DBMSs provide consistency and availability, cloud storage systems provide availability and partition tolerance. This is done by relaxing the ACID constraints in cloud storage systems as it significantly reduces the communication overhead in a distributed system, providing simple data access APIs instead of complex query interfaces, and schema-free data storage [Bur10].

Another category of cloud storage systems are distributed filesystems. Distributed filesystems in general have been available for over 30 years, their purpose is to “allow

users of physically distributed computers to share data and storage resources by using a common file system" [LS90]. Well-cited examples are Sun's Network File System (NFS) [SGK+88] and the Sprite Network File System [OCD+88]. However, a distributed file system—such as the Google File System—which is designed to act as a cloud storage system, differs from traditional distributed filesystems in a similar way as a distributed key-value storage differs from a relational database management system. It weakens consistency, reduces synchronization operations along with the introduction of replicating data multiple times in order to scale gracefully and provide high-availability. This can be illustrated by reviewing the underlying assumptions and requirements that led to the design of the Google File System [GGL03]: the file system is built from many inexpensive commodity components where hardware failures are the norm not the exception, which means that fault-tolerance and auto-recovery need to be built into the system. Files can be huge, bandwidth is more important than latency, reads are mostly sequential and writes are dominated by append operations. The corresponding Google File System architecture foresees that a GFS cluster has a single master server, multiple chunkservers and they are accessed by multiple clients. Files in GFS are divided into fixed-size chunks, where each chunk is identified by a unique 64-bit chunk handle, the size of a chunk is 64 MB, and chunks are replicated at least three times throughout the cluster. A master server maintains all file system metadata such as namespaces, access control information, the mappings from files to chunks as well as the locations from the different chunks on the chunkservers. Each chunkserver stores its chunks as separate files in a Linux file system. If a client wants to access a file, it contacts the master server which provides it with the chunk server locations as well as the file-to-chunk mappings: the client is then enabled to autonomously contact the chunkservers. This allows clients to read, write, and append records in parallel at the price of a relaxed consistency model (e.g. it may take some time until updates are perpetuated to all replicas): these relaxed consistency guarantees have to be covered by the applications that run on top of GFS (e.g. by application-level checkpointing).

With the increasing popularity of cloud storage systems emerged the need to execute more complex operations on the stored data than simple data retrieval and modification operations. This need is addressed by Google's MapReduce programming model [DG08] that was built for being used in conjunction with Google Bigtable or the Google File System. The idea of MapReduce is to process large sets of key-value pairs with a parallel, distributed algorithm on a cluster: the algorithm performs at first a Map() operation that performs filter operations on key-value pairs and creates corresponding intermediate results followed by a Reduce() operation which groups the intermediate results foe example by combining all results that share the same key. The simplicity of this approach, the ability to quickly identify non-interleaving data partitions, and the ability to execute the respective sub-operations independently, enable a great degree of parallelism.

2.4.1 State-of-the-Art Cloud Storage Systems

Summary: This subsection presents state-of-the-art cloud storage systems.

After describing Amazon Dynamo, Google Bigtable, and the Google File System in the previous subsection, we discuss additional state-of-the-art cloud storage systems.

- **Amazon Dynamo** [DHJ+07] see previous Sect. 2.4.
- **Google Bigtable** [CDG+06] see previous Sect. 2.4.
- **Google File System** [GGL03] see previous Sect. 2.4.
- **Hadoop** [Whi09] and the **Hadoop Distributed File System** (HDFS) [SKRC10] are open-source implementations based on the concepts of Google's MapReduce and the Google File System. HDFS has a similar architecture as GFS as it deploys a central entity called namenode that maintains all the meta information about the HDFS cluster. So called datanodes hold the data which gets replicated across racks, and clients that directly interact with datanodes after retrieving the needed metadata from the namenode. HDFS differs from GFS in the details (e.g. HDFS uses 128 MB blocks instead of GFS's 64 MB chunks). In HDFS a client can freely choose against which datanode it wants to write, and HDFS is aware of the concept of a datacenter rack when it comes to data balancing. Hadoop itself is a framework for executing MapReduce, it includes and utilizes HDFS for storing data, and provides additional modules such as the MapReduce engine: this engine takes care of scheduling and executing MapReduce jobs, and consists of a JobTracker which accepts and dispatches MapReduce jobs from clients and several TaskTrackers which aim to execute the individual jobs as close to the data as possible.
- **Memcached** [Fit04] is a distributed key-value store which is commonly used for caching data in the context of large web applications (e.g. Facebook [NFG+13]). As a consequence, each server in a Memcached cluster keeps its data resident in main memory for performance improvements. Upon a power or hardware failure, the main memory resident data is lost. However, this is not considered harmful as it is cached data, which might be invalidated after potential server recovery. Memcached itself has no support for data recovery, it is expected to be provided by the application (e.g. Memcached is extensively being used at Facebook and the Facebook engineering implemented their own data replication mechanism [NFG+13]). The data in a Memcached cluster is partitioned across servers based on hash values of the to be stored keys: their ranges are mapped to buckets and each server is assigned one or more buckets.
- **Project Voldemort** [SKG+12] is a distributed key-value store developed by the social network LinkedIn and is conceptionally similar to Amazon's Dynamo. Project Voldemort also applies consistent hashing to partition its data across nodes and to replicate data over multiple times with a configurable replication factor. Project Voldemort also does not provide strong consistency, but facilitates a versioning system to ensure that data replicas become consistent at some point.
- **Stanford's RAMCloud** [OAE+11] is a research project that combines the in-memory performance of a solution such as Memcached with the durable,

high-available, and gracefully scaling storage of data as realized by a project such as Bigtable. It does so by keeping all data entirely in DRAM by aggregating the main memory of multiple of commodity servers at scale. In addition, all of these servers are connected via a high-end network such as InfiniBand (as discussed in Sect. 2.1.3) which provides low latency [ROS+11] and a high bandwidth. RAMCloud employs randomized techniques to manage the system in a scalable and decentralized fashion and is based on a key-value data model. RAMCloud scatters backup data across hundreds or thousands of disks or SSDs, and harnesses hundreds of servers in parallel to reconstruct lost data. The system uses a log-structured approach for all its data, in DRAM as well as on disk/SSD, which provides high performance both during normal operation and during recovery [ORS+11]. The inner workings are explained in detail in Sect. 3.3.

2.4.2 *Combining Database Management and Cloud Storage Systems*

Summary: This subsection discusses different approaches of combining database management and cloud storage systems, including the adaptation of each other's features, providing connectors, translating SQL to MapReduce programs, providing specialized SQL engines on top of cloud storage systems, having a hybrid SQL/MapReduce execution, and utilizing a cloud storage system as shared-storage for a DBMS.

The advent of cloud storage systems piqued interest in the DBMS as well as in the cloud storage systems community to evaluate the use and adaptation of each other's features. Initially, this was an unstructured undertaking which for example resulted in the statement by Michael Carey that "it is the wild west out there again" [ACC+10]. Dean Jacobs said "I recently reviewed a large number of 'cloud database' papers for various conferences. Most of these papers were either adding features to distributed key-value stores to make them more usable or removing features from conventional relational databases to make them more scalable" [Jac13]. However, the adaptation of relational query processing features in a cloud storage system eventually became a well-established area of work in academia as demonstrated by the Stratosphere project [Mem13b], which extends the MapReduce model with operators [BEH+10] which are common in relational DBMS.

The advantages and growing popularity of cloud storage systems led to the desire to execute SQL statements against data that is inside a cloud storage system. The different approaches can be put into the following four categories:

- **DBMS to cloud storage system connectors** allow the bidirectional exchange of data between the two systems. Such an approach is commonly used for running ad-hoc queries on the outcome of a MapReduce job by preparing the unstructured data in the cloud storage system and then convert it to structured data inside the DBMS. Those connectors are popular with traditional DBMS vendors as they allow them

to label their products as "Hadoop compatible". Examples for such connectors are the Microsoft SQL Server Connector for Apache Hadoop [Cor13] and the HP Vertica Hadoop Distributed File System Connector [Ver13]. The disadvantages of such connectors are that they (a) require an ETL process which forbids ad-hoc querying, (b) transfer the complete to be queried dataset over the network, and (c) create a redundant copy of the dataset.

- **SQL translated to MapReduce** allows sending an SQL query to a cloud storage system such as Hadoop. The SQL query is translated to a set of MapReduce jobs which are then executed by the cluster. This bears the advantages that it (a) utilizes the properties of the cloud storage system in terms of high-availability as well as scalability and (b) integrates into the existing scheduling of MapReduce jobs (as opposed to the previous approach where the extraction of data creates an unexpected extra load). The main disadvantages are that (a) the accepted SQL is a SQL dialect and not SQL-standard conform and (b) the translation overhead and the MapReduce batch-oriented execution style prevent ad-hoc queries. Facebook's Hive [TSJ+09] is an example of a system that uses such an approach.
- **Specialized SQL engines on top of cloud storage systems** accept SQL-like queries. They execute the queries not by translating them to MapReduce jobs, but by shipping custom database operators to the data nodes. Systems that fall into that category are the row-oriented Google F1 [SOE+12] and the column-oriented Google Dremel [MGL+10]. For example Google Dremel operates on GFS and exploits the GFS interfaces that allows code execution on chunkservers and thereby to ship and execute operators. This results in the advantage of being able to execute ad-hoc queries as well as collocating data and their processing. The big disadvantages of such an approach are that (a) such specialized SQL engines are not SQL-standard compliant and (b) they only provide poor coverage of the common SQL operators. These disadvantages eliminate the use of existing tools and the ability to execute applications which expose SQL-standard compliant queries.
- **Hybrid SQL and MapReduce execution** aims at combining both of the previous approaches: an SQL query submitted to the system gets analyzed and then parts of it are processed via MapReduce and other parts with the execution of native database operators. That allows to determine the mix of MapReduce and database operator execution based on the type of query: for executing ad-hoc queries MapReduce-style execution is avoided as much as possible whereas it is preferred for queries at massive scale in combination with the need for fault tolerance. Examples for such systems are HadoopDB [BPASP11] or Polybase [DHN+13].

Another category of research that focuses on the combination of database management and cloud storage systems is the **use of cloud storage systems as shared-storage for parallel DBMS**. Whereas the previously explained approaches trim the SQL coverage and sacrifice the compliance with the SQL standard, this approach takes a standard relational query processor or DBMS and utilizes the cloud storage system instead of a classic shared-disk storage.

- **Building a Database on S3** [BFG+08] by Brantner et al. demonstrates the use of Amazon S3 as a shared-disk for persisting the data from a MySQL database. This work maps the elements from the MySQL B-tree to key-value objects and provides a corresponding custom MySQL storage engine that allows for prototypical experiments. It also introduces a set of protocols which show how different levels of consistency can be implemented using S3. Driven by the TPC-W benchmark, the trade-offs between performance, response time, and costs (in terms of US dollars) are discussed.
- **Running a transactional Database on top of RAMCloud** [Pil12] by Pilman takes a similar conceptual approach, but utilizes RAMCloud instead of Amazon S3. The motivation for using RAMCloud is to exploit its performance advantages provided by in-memory data storage and RDMA capabilities. The work by Pilman presents two different architectures: one where a MySQL instance runs exclusively on a RAMCloud cluster, and the other one where several instances of MySQL run on a RAMCloud cluster. A benchmark is presented that executes TPC-W and uses MySQL with InnoDB as a baseline. The experiments show that "we can run MySQL on top of a key-value store without any loss of performance or scalability but still gain the advantages this architecture provides. We have the desired elasticity and several applications could run in the same network using each its own database system, but all on the same key-value store" [Pil12].

2.5 Classification

Summary: This section classifies related work as presented throughout the chapter.

After an extensive explanation of the background and the related work in the previous sections of this chapter, this section presents a compact overview on the related work and the corresponding classification.

This work is positioned in the field of evaluating a parallel DBMS architecture and its implications on query processing. As explained in Sect. 2.3.1, the shared-nothing versus shared storage architecture trade-offs are much discussed in the context of classic storage architectures (e.g. SAN/NAS storage) and a great variety of products from big vendors are available in both markets. It is noteworthy that for main memory resident parallel DBMSs a shared-nothing architecture is dominating: this is due to the intention of not sacrificing the performance advantage of keeping data in main memory, but constantly shipping it over a significantly slower network. However, with the advent of fast switch fabric communication links—as discussed in Sect. 2.1.3—the performance gap between accessing local and remote main memory narrows down and the implications of this development on the architecture discussion for main memory parallel DBMS are not clear yet.

Deploying a parallel DBMS on a cloud storage system is a relatively new area of research. As explained in Sect. 2.4.2, the cloud community has worked out several approaches how to execute SQL-like statements against data in a cloud storage system. Specialized SQL engines on top of cloud storage systems most closely resem-

ble a traditional DBMS as they just use database operators for query execution and neglect the batch-oriented MapReduce paradigm altogether. But even those systems are not SQL-standard compliant and provide their own SQL dialect which makes them uninteresting for the broad range of applications that expose SQL-standard compliant queries. This downside is not inherent to the work from the DBMS community which takes the opposite approach by deploying a standard, SQL-standard compliant DBMS on to a cloud storage system. In this field, the work co-authored [BFG+08] and supervised by [Pil12] Donald Kossmann are the single most related pieces of work.

In this work, we also focus on deploying a parallel DBMS on a cloud storage system, but (a) we keep all data resident in main memory all the time, (b) we apply a column-oriented data layout, and (c) we use the storage system for both—data access and code execution. This area of work in currently not addressed in the research community. Google works with Dremel in the same domain, but their approach is based on disk resident data. So far, the work co-authored [BFG+08] and supervised by [Pil12] Donald Kossmann, focuses on the processing of transactional workloads by a row-oriented database and it utilizes the cloud storage system solely as passive storage without considering the possibilities of operator shipping and execution.

http://www.springer.com/978-3-319-20710-0

Building a Columnar Database on RAMCloud
Database Design for the Low-Latency Enabled Data Center
Tinnefeld, C.
2016, XIX, 130 p. 37 illus., Hardcover
ISBN: 978-3-319-20710-0