A Survey on Parallel Database Systems from a Storage Perspective: Rows versus Columns

Carlos Ordonez[1] * and Ladjel Bellatreche[2]

[1] University of Houston, USA
[2] LIAS/ISAE-ENSMA, France

Abstract. Big data requirements have revolutionized database technology, bringing many innovative and revamped DBMSs to process transactional (OLTP) or demanding query workloads (cubes, exploration, preprocessing). Parallel and main memory processing have become important features to exploit new hardware and cope with data volume. With such landscape in mind, we present a survey comparing modern row and columnar DBMSs, contrasting their ability to write data (storage mechanisms, transaction processing, batch loading, enforcing ACID) and their ability to read data (query processing, physical operators, sequential vs parallel). We provide a unifying view of alternative storage mechanisms, database algorithms and query optimizations used across diverse DBMSs. We contrast the architecture and processing of a parallel DBMS with an HPC system. We cover the full spectrum of subsystems going from storage to query processing. We consider parallel processing and the impact of much larger RAM, which brings back main-memory databases. We then discuss important parallel aspects including speedup, sequential bottlenecks, data redistribution, high speed networks, main memory processing with larger RAM and fault-tolerance at query processing time. We outline an agenda for future research.

1 Introduction

Parallel processing is central in big data due to large data volume and the need to process data faster. Parallel DBMSs [15, 13] and the Hadoop eco-system [30] are currently two competing technologies to analyze big data, both based on automatic data-based parallelism on a shared-nothing architecture. On the other hand, larger memory capacity has triggered rearchitecting systems to push more processing to main memory. Nowadays the major DBMS storage technologies are rows and columns. Rows are the most common storage mechanism, researched for decades. They provide great performance for common OLTP queries, but can be slow to process complex queries combining joins and aggregations. Yet during the past decade columnar database systems (i.e. DBMSs using column-based storage) [1, 18, 31] have become a major alternative to compute complex SQL queries on large databases, providing at least an order of magnitude in performance improvement compared to row-based DBMSs [31, 33], and two orders of magnitude

* Work partially conducted while the first author was visiting MIT.

compared to the Hadoop ecosystem (MapReduce [9], Spark [36]). Unfortunately, column storage requires rewriting many subsystems from the ground up. With that motivation in mind, we present a survey on parallel DBMSs, covering the state of the art and issues for future research. We consider the implications of row and columnar storage, highlighting which processing tasks are easier or more difficult on either storage mechanism. Since it is a major benchmarking effort to compare all systems on loading speed, ACID compliance, query expressiveness and query processing speed we do not include any experiments. Instead we aim to identify main system architecture characteristics, storage mechanisms, and time complexity of algorithms evaluating physical operators.

We provide a unifying view and classification of storage mechanisms, query processing algorithms, novel optimizations and parallel processing across different systems. We cover the full spectrum of subsystems going from storage to analytic processing, contrasting columnar and row DBMSs. Based on the state of the art, we outline a vision of research issues. Due to lack of space we omit many references, especially commercial systems (e.g. white papers, web sites).

2 Preliminaries

Notation: To characterize time and space complexity, we use n to denote table size (number of records), p as a variable for the number of columns (of diverse data types) and P for the number of nodes/processors in a parallel system.
DBMS: We focus on parallel DBMSs, which we classify as row or columnar. A DBMS has the following distinguishing features comparared to other large software systems: storage is available only for structured data: relational tables (most common), but also disk-based arrays (new trend); a table schema must be defined before storing data (inserting new records or loading them in batch), querying (in general with SQL, but also with alternative languages like Datalog [2], or XML [13]) and maintaining ACID properties (atomic, consistent, isolated, durable) to maintain data in a complete & consistent state [13].
Landscape: Row DBMSs represent a mature technology dating back several decades [13], pioneered by IBM System R, then followed by Oracle, SQL Server and Teradata. Columnar DBMSs came one decade ago to tackle complex SQL queries. Influential columnar DBMSs include Sybase [24] (a pioneer in columnar storage), MonetDB [18, 25] (industrialized as VectorWise) and C-store [1, 31] (rewritten and commercialized as Vertica [22, 34]). Other columnar DBMSs worth mentioning are SAP Hana [12] (main memory columnas DBMS, integrated with SAP), Virtuoso. Finally, columnar storage was grafted into older row-based DBMSs like SQL Server (in the form of indexes), Teradata (as user-defined table or precomputed join index) and Oracle (tables in main memory). There is a third class of DBMS with innovative storage: arrays, enabling unlimited size multidimensional arrays and matrices, fundamental in machine learning (SciDB [32], Rasdaman [5]). Due to big data requirements and open-source trends, DBMSs now feature automated schema requirements and enable fast data transfer with Big Data systems. But at the same time "Big Data" Hadoop systems (working on

HDFS) have evolved to support SQL queries and more recently transactions with varying ACID guarantees. That is, they have evolved, getting closer to relational DBMSs. A major difference is that DBMSs are generally monolithic, whereas open-source Hadoop systems tend to be modular allowing different subsystems being assembled together (the so-called Hadoop stack). There exist alternative Big Data database systems and subsystems with weaker OLTP features (weaker no ACID properties), noSQL (alternative query languages), generally built on top of the Hadoop distributed file system (HDFS). Figure 1 shows a taxonomy of Big Data Analytics systems.

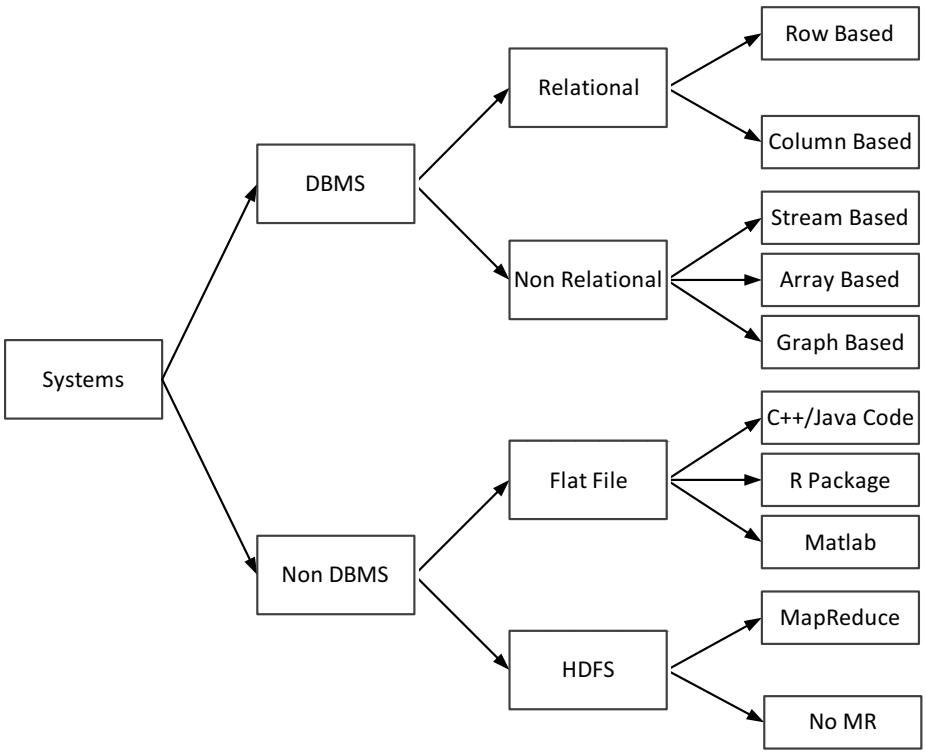

Fig. 1. Classification of Parallel DBMSs and Big Data Systems based on storage.

3 Parallel System Architectures

There exist diverse parallel system architectures, going from a multicore CPU to multiple CPUs interconnected with a high speed network. The common goal is to exploit as many processing units as possible; such processing units can be multiple threads running on one CPU, several cores on multicore CPUs, many cores

in a GPU or a set of servers running on a cluster. Most DBMSs have a shared-nothing parallel architecture with independent secondary storage, whereas HPC systems allow more interconnectivity between processors and secondary storage. Another major difference is the inter-unit communication: HPC systems favor MPI, whereas parallel DBMSs prefer plain UDP sockets. The main reason is intensive I/O on a shared disk (or disk array) or shared memory accessed by multiple threads become a bottleneck. These days there is a divide in HPC between having one machine with many cores and large shared RAM or a cluster with independent RAM and a fast node interconnect. In short, we assume a shared-nothing parallel DBMS architecture with P processing nodes, each with its own memory and disk [10, 30, 33]. A second major difference is main memory access. In general, HPC systems load as much data as possible into main memory across the cluster in one phase, whereas parallel DBMSs use a buffer (a.k.a. cache) with clever page eviction. Given larger RAM there is a renewed trend to avoid I/O and perform most transaction and query processing in main memory.

4 Writing: Storage and Updating Database

We contrast row and columnar DBMSs, highlighting internal system changes. We justify the need to define a schema. For each subsystem we will first present a unifying view of current research, identifying common mechanisms, algorithms, data structures and optimizations, emphasizing tradeoffs among them and pointing out optimizations not used in practice.

Table 1 provides a summary of popular parallel systems. It shows DBMSs supporting SQL, HDFS systems and alternative systems that offer some DBMS functionality. This table is by no means complete as it omits many smaller systems with more specific functionality, or tailored to specific analytic problems.

4.1 Storage mechanisms

Files: In a row DBMS running on one machine each table is generally stored on one file, partitioned into blocks of rows (unless it is "sharded"). Thus reading a row implies reading all columns. In contrast, with column-based storage queries read only the few columns they need from a wide denormalized table [31]. At the logical level columnar DBMSs exploit relational projections [23, 22, 18] of the form $\pi_{i,A_1,A_2,...,A_p}(T)$ on table T, where i is an internal row identifier. In general, such projections do not represent materialized views because they do not involve joins and aggregations [18, 22, 24, 31]. The main optimization is to process queries on such projections instead of T. A well chosen set of projections can substitute T [31], although it is preferable to materialize T anyway [22]. At the physical level each projection is further partitioned into columns, where values for each column A_j are stored in a separate file. An essential internal reference mechanism is that the row id i is not stored, but it is an implicit array subscript based on some value ordering [1]. Such approach has two fundamental benefits: (1) it decreases storage space [1] and it reduces I/O cost [18]; (2) it

Table 1. DBMSs & Big Data systems: storage, query language, parallel architecture.

Name	Storage	Query Language	Shared-nothing P nodes
DataDepot	Row	SQL	Y
DB2	Row	SQL:2011	Y
Greenplum	Row	SQL:2008	Y
MemSQL	Row	SQL	Y
MySQL	Row	SQL:1999	Y, sharding
Oracle	Row	SQL:2008	Y
SQL Server	Row	Transact SQL	N, Y in PDW
PostgreSQL	Row	SQL:2008	N
SQLite	Row	SQL	N
Teradata	Row	SQL:1999 and T-SQL	Y
VoltDB	Row	SQL	Y
InfiniDB	Columnar	SQL:92	Y
MonetDB	Columnar	SQL:2008, SciSQL	N
SAP Hana	Columnar	SQL:92	Y
Vertica	Columnar	SQL:1999	Y
Accumulo	HDFS	Thrift	Y
Hbase	HDFS	(J)Ruby's IRB	Y
Impala	HDFS	SQL:92	Y
Shark	HDFS	SQL	
PostGIS	Array/block	SQL:2008	N
Rasdaman	Array/raster	RaSQL	N
SciDB	Array/chunk	AQL	Y

allows using i as an array subscript, bypassing indexing mechanisms [1, 18]. As a consequence, the boundary between the logical and physical storage levels gets blurred, in contrast to row DBMSs. In general, the row identifier i is derived from the column value position inside a block. Column values are generally stored in compressed form; in such case the system determines an offset by multiplying the value position by the value frequency. Storing columns in separate files allows multithreaded parallel reads and it yields high compression ratios [31, 18], but it requires assembling rows (materialization) at the end of query processing. Array DBMSs have many similarities with columnar DBMSs: they store each attribute in separate storage units, they partition arrays by column, they feature improved locality, larger blocks and compression. The main differences are that multidimensional arrays have uniform content across their dimensions and they need mathematical and physical operators incompatible with relational algebra.

I/O unit: In row DBMSs a block of rows remains the I/O standard. Compared to row OLTP systems a larger I/O unit is more common in columnar systems. This generally results in a big logical block: segment in a columnar DBMS [22]. The rationale is that a larger block favors long scans and minimizes seeks for query processing. Blocks may have non uniform size depending on value distributions and compression. Grouping column values into blocks requires considering two aspects [18, 22, 31]: improving locality of access and parallel processing. The system improves access locality by maintaining values sorted. Fixed size BLOBs (pure byte strings) enable direct loading and array access [18, 23, 37], but they are not compatible with compression.

Hybrid storage mixing rows and columns: Can rows and columns coexist in the same storage system? yes, but it is difficult because the physical operator executor requires significant changes. Can we store row and columns on the

same block? There is not a clear answer. The net result: performance not good enough compared to a pure column store [1]. Since full support for column storage requires rewriting the storage manager from scratch, legacy row DBMSs have opted for limited column optimizations [23]. Currently, the fastest columnar DBMSs have two separate internal storage managers [12, 22], as we explain later.

4.2 Fast Access: Ordering Rows versus Indexes

Ordered value storage to avoid indexing: There are two major alternatives to improve access time to a subset of rows based on a search key: adding indexes to improve search time (row DBMS) or maintaining sorted projections (columnar DBMS). In row DBMSs the most common data structures to index column values are B-trees and hash tables with $O(\log(n))$ and $O(1)$, average time respectively. In a columnar DBMS since query processing requires searching values there are two major alternatives to decrease search time: maintaining values sorted or using an auxiliary data structure to search values. If values are sorted search takes time $O(\log(n))$. But since an index is a projection of a table most approaches favor maintaining values sorted, which can be done in an efficient manner when insertions come in batches. Notice ordering is crucial for time series, spatial data and streams: queries tend to access value ranges.

Compression: Both row DBMSs and columnar DBMSs exploit compression, but they sharply differ on how they process queries. Two reasons motivate compression: the existence of a few frequent column values and maintaining column values physically sorted. We emphasize compression of columns with repeated values is not a new idea, since compression is commonly applied on row DBMSs and compression is a common mechanism to accelerate data transfer in networking. Moreover, efficiently storing long sequences of repeated symbols is well known in text processing. Columnar storage provide much better compression than row stores [1]. In fact, a subtle aspect that is not well known is that columnar DBMSs actually provide higher compression rates than popular compressing utilities [22] (e.g. gzip, winzip). Thus compression significantly reduces I/O cost and network traffic in a parallel system. Yet the overhead to decompress values at query time must be considered. Since storage is column based, the chosen compression granularity is per column, which creates blocks of compressed data. The two most common compression mechanisms in columnar DBMSs are run-length encoding (RLE) and dictionary encoding (DE), with RLE being the most popular. RLE can only be applied when values are sorted and then counted. In an alternative manner, DE is preferred on row DBMSs and it is useful when there are few frequent column values; this scheme is more effective if values take many bytes (i.e. long strings). Hash tables and arrays are two common mechanisms to program dictionaries with a code of a fixed bit length. There exist other compression mechanisms like delta encoding, with less general applicability. Compression mechanisms that produce variable length codes or that encrypt data are slower at query time. So LZW & Huffman codes or arithmetic compression are a bad idea in a DBMS [1, 31]. In short, lightweight compres-

sion is preferable to enable faster decompression [22, 37] or working directly on a compressed data representation [31, 22].
Indexing: In a row DBMS B-trees are preferred for point/range queries and transaction processing; hashing is preferred to distribute rows for parallel processing and accelerate join computation; bitmaps are preferred to accelerate "count" aggregations. In contrast to row DBMSs, a columnar DBMS [22] does not use row-level indexing; sparse indexes (trees on min/max values per compressed block [31]) are eagerly created when the user defines projections [22] or dynamically by the DBMS during query processing (database cracking [18]). Some row DBMSs expose columnar projections as special indexes [23].

4.3 Inserting and Updating Database Records

We consider two major mechanisms to update the database: inserting/updating a few records, most commonly done by transactions and loading a batch of records, generally done in a data warehouse or analytical database. Such mechanisms lead to two prominent concurrent processing scenarios: (1) OLTP: Updating or deleting few records concurrently with multi-threaded processing in shared memory or (2) OLAP/cubes: doing batch loading and processing complex queries at the same time. We discuss each mechanism below.
Transactions: Transactions (small SQL programs updating the database) enable concurrent updates and provide fault tolerance. Row DBMSs remain the best technology to process transactions, where locking (2PL, optimistic [13]) and multi-version concurrency control (MVCC+timestamping) [13] remain the most common mechanisms to enforce ACID properties (i.e. serializable and recoverable schedules). It is generally agreed locking is slower but provides strongest consistency guarantees, whereas MVCC is faster but creates multiple inconsistent views of the database. Recently row DBMSs have started moving towards lock-free approaches using timestamping [33] exploiting main memory processing with single threaded processing (very fast, significantly simplifying OLTP processing and recovery, defying old assumptions). A main reason behind such acceleration is that they exploit servers with much larger RAM. When performing parallel processing in a cluster timestamping can be used on multiple nodes if the transaction data records can be partitioned and routed for local processing. For small and medium size OLTP databases (relative to large RAM) it is feasible to maintain the database in main memory (e.g. VoltDB) across all servers in the parallel cluster. On the other hand, transaction processing is considered infeasible in a columnar DBMS, because to update a row columns should be decompressed, rows dinamically assembled and then compressed again and written back. Moreover, updating values on multiple rows (especially non-key columns) in a columnar DBMS is significantly slower than row DBMSs due to the need to decompress and compress and potential data re-partitioning [31] (a.k.a. shuffling). As explained below, a common solution is to cache recent updates in batch in a data structure in RAM [12, 22]. In general, columnar DBMSs provide minimal transaction processing features, but provide reasonable (but weaker) ACID gurantees [12, 22]. On the other hand, to enable fast bulk insertions (no

updates or deletes) and concurrent querying most systems provide snapshot isolation, internally managed with record versioning. That is, records are inserted into blocks that cannot be accessed by the user until loading is complete.
Loading data: Loading data from row OLTP databases into a central row database (data warehouse) is fast and in some systems (Teradata, MemSQL) is done in near real time (e.g. active data warehousing). In a columnar DBMS, loading data in batch is generally a bottleneck because row to column transformation is required [18, 22], which in turn requires sorting each column. Thus transforming row by row to columns is slow [22, 31] (akin to matrix transposition). A solution is having two internal storage managers with a batched tuple mover between a write-optimized row store and a write-optimized column store. In general, the write-optimized store works in RAM [12, 22], where refreshing data with δ tuples is done as a background process to avoid query interruption [22]. Sybase IQ [24] explicitly forces a tuple order, by time, to avoid sorts.

5 Reading: Query Processing

We first discuss how sequential processing happens on one machine or one thread. Based on such foundation, we then discuss how sequential processing is generalized and extended to work in parallel.

5.1 Sequential Query Processing

Physical operators: In a row DBMS, there are scan (read all rows), join (merge-sort, hash), sort and search (indexed or sequential) algorithms. On the other hand, in a columnar DBMS, there are newer versions of scan, join, sort and search algorithms acting at the column level [20]. A scan operator brings blocks directly into RAM. These are the major scan variations: reading small blocks that can fit in the CPU cache memory [18], reading medium blocks with compressed column values [22], reading large blocks with uncompressed column values. Scans on multiple columns are optimal when such columns are aligned on the same order, enabling direct manipulation with arrays. There are two preferred kinds of join algorithms: merge joins and hash joins [18, 22, 31], where merge joins avoid the sort phase in the classical sort-merge join. In general, most DBMSs avoid nested loop joins. In contrast to row DBMSs, merge joins are preferred over hash joins, when input tables have their projections already sorted by the joining columns. Otherwise, DBMSs use hash joins as default, creating a hash table on the (inner) smaller participating table [22]. When both tables are compressed on the same key and merge join is feasible time complexity is $O(k)$, where $k = |\pi(K)|$ for the join key K. Sorting is necessary when no projection satisfies a join or aggregation GROUP BY required order or when the cardinality of the column is large. Sorting (including external sort) generally uses the traditional merge sort algorithm in time $O(n\log(n))$ to create a sorted projection. Sequential searches are avoided. Whenever there exists a projection physically sorted by a key the default search algorithm is binary search, which can be as

efficient as $O(\log(k))$ on a compressed column, or $O(log(n))$ otherwise. Otherwise, time is $O(n)$, but assumed a rare occurrence. In a row DBMS projection π takes time $O(pn)$, regardless of how many columns are projected. In contrast, in a columnar DBMS projection takes time $O(mk)$ for m projected columns and k distinct values, and in most cases $m \ll p$ and $k \ll n$.

Query executor: To evaluate each operator a row DBMS takes as input rows from one or two tables (for unary and binary operators, perhaps pipelined) and produces rows on one output table. A columnar DBMS is significantly different, having these main steps: determine required columns, read column values by block, search column blocks (exploiting some indexing data structure), transfer blocks to buffer in RAM, process blocks in RAM with column physical operators, and assemble the rows at some point. Processing algorithms across DBMSs vary significantly depending on compression. If blocks are compressed it is preferable to process them in compressed form, delaying decompression until results are returned [1]. Dictionary encoding does not require decompressing in intermediate evaluation steps, whereas run-length encoding helps query evaluation, by directly processing the repeated value and its frequency. Queries require eventually assembling rows from column values. There are two major approaches to assemble rows: early materialization [18] and late materialization [22, 31], leading to significantly different query processing. Late materialization provides best performance when compression is heavily used [1]. The level of integration of the query processor with the hardware varies significantly going from a small query optimizer/processor kernel, tightly integrated with the operating system [18] to a fully fledged big optimizer [22], comparable to legacy DBMSs.

Query plan: Most DBMSs provide ANSI SQL compatibility. Most row DBMSs represent the query plan in a similar manner, following IBM System R [13]. In contrast, columnar DBMSs differ significantly on how queries are internally represented and optimized. If a traditional relational algebra approach is used, relational query transformation rules and cost-based query optimization are applied together [22, 31]. On the other hand, other DBMSs define a simpler algebra on narrow projections (e.g. binary association tables [18, 37]). to enable tight integration with the CPU and operating system and a smaller space for potential query plans (operator commutativity is more limited). Query transformation favors pushing projection instead of pushing selection in SPJA queries [1, 22]. In contrast to row DBMSs, when projections share ordering by the same key, join operators are evaluated first since they act as a filter. In a query optimizer with full place space exploration, like [22], query trees tend to be bushier [22] instead of left deep [13]. Thus plan space exploration becomes harder with bushy trees . Some DBMSs use traditional cost models combining CPU, disk I/O and network transfer cost [22], with more upfront optimization, whereas other DBMSs delay such optimization, combining query transformation with dynamic storage re-organization and adaptive indexing [18, 37].

Non-relational operators and functions: It is no surprise there is work on adapting DBMSs to analyze non-relational data [18]. The two most outstanding novel data types are streams and arrays. Stream data records arrive row by

row [16]: they require a different query processor combining main memory and secondary storage. Also, streams arrive (mostly) in time order which complicates reorganizing storage when accessing and ordering by a different column. In order to keep up with stream speed column stores require a fast incremental converter of new rows. A common form of stream operator has a stream in RAM and a table on disk (i.e. a join between a table and a stream). On the array angle, unfortunately arrays and relational tables are incompatible with each other. Therefore, it is necessary to develop new operators (sometimes extending SQL) and new languages that can manipulate arrays.

5.2 Parallel Query Processing

Data partitioning: This is statically done at table loading or dynamically during query processing [13, 6]. Sharding [30] creates subsets of tables and can work across new servers. To provide 24/7 availability, it is a requirement to add/remove hardware (including nodes), without disruption. Such dynamic hardware expansion is particularly popular with MapReduce (MR) [9, 21], Spark [36] and specialized systems on top of HDFS [3]), which exploit heartbeats, data replication and daemon processes. Data redistribution (shuffling) happens when required rows or values are not available on the same node, transferring data to different nodes. There are two major approaches to partition tables across P nodes: hashing or range-based, like row DBMSs [13, 22]. Parallel DBMSs provide three basic operators to send and receive data blocks [13, 22]: (1) broadcast a block to all P nodes (scatter). (2) converge-cast from P nodes to 1 node (gather). (3) node to node 1-1. There are two partitioning levels: inter-node horizontal partitioning and intra-node partitioning (which extends horizontal partitioning). Data redistribution happens after some physical operators and it cannot be guaranteed results will remain on the same node. So load re-balancing is required. Sharding [30] is another important partitioning technique, which horizontally partitions tables into "focused" subsets to avoid full table scans when analyzing data.

Parallel speedup: We compare merits and limitations of each approach with respect to parallel processing theory [11]. Currently, there are two parallelism dimensions: scale-up and scale-out. Scale-up is represented by multicore CPUs, GPUs using multi-threaded processing on a single node [18], where main memory may be shared or partitioned. On the other hand, scale-out is represented by multi-node parallel processing. Multi-core CPUs and GPUs keep growing the number of cores, but their speed has reached a threshold. GPUs represent an alternative with a much higher number of cores, but with separate main memory and different API. RAM keeps growing, but multiple CPUs compete for it (i.e. contention for a shared resource). Network transfer remains slower than the transfer from CPU to RAM, although fast CPU interconnections are changing the landscape. Therefore, in general network communication is the bottleneck, followed by RAM (i.e. the memory wall [18]). All systems aim to achieve linear speedup, minimizing the sequential bottleneck (Amdhal's law [11]) and communication overhead. The parallel cluster topology (seen as a graph connecting P

nodes) can be: complete graph ($O(P^2)$ connections), tree (balanced binary tree $O(P)$ connections), grid (connecting CPUs and disks). Most DBMSs assume a complete graph to enable point-to-point data transfer, but synchronized transfer. Network communication is commonly done with sockets (medium speed), with MPI (slower, but with higher reliability, compatible with HPC applications) or with high-speed interconnect hardware (e.g. Infiniband). MonetDB [18] (commercially VectorWise) is the best DBMS exploiting hardware: multicore CPUs, increasing cache hits; exploit locality of reference, exploiting faster seek on flash, fast long scans on disk and flash.

Parallel physical operators: There are parallel versions of scan, join and sort [18, 22, 31, 37]. Scan works by transferring compressed blocks to the requesting node, decompressing at the end. The fastest join algorithms (highlight differences) include hash joins or merge joins. Nested loop joins are rare, but they are used by replicating a small table at all the P nodes to be joined with a much larger table. The default sorting algorithm is merge-sort. In general, sorting an entire table happens only during query processing because insertion maintains table columns sorted separately. Big blocks minimize network traffic as long as all values are relevant (i.e. pre-filtered and with a useful value ordering).

Fault tolerance: Parallel processing at massive scale requires fault tolerance to process long lasting queries and graph/ML analytics, to avoid restarting jobs [22, 30]. Thus, this is similar to MapReduce and Spark. Some DBMSs provide K-safety [31, 22] in which $K + 1$ copies must be maintained to tolerate up to K node failures. To guarantee safe operation, at any time A, the number of active nodes, must be $A > P/2$.

6 Conclusions: a Tentative Agenda for Future Research

Row DBMSs are best for transaction processing, very fast for batch loading and provide good performance for complex queries, whereas columnar DBMSs are bad for transaction processing, reasonably fast for batch loading and fastest for complex queries. Even so there are many opportunities for future research. Big Data Analytics is an emerging area, going beyond database systems. Here we categorize important research problems into core database research, where DBMS technology prevails and where problems are well established and big data analytics systems, where the norm today is cloud computing and a weak enforcement of a database schema. Due to space limitations, we omit discussion on database modeling (ER model, process management), data pre-processing (data cleaning, ETL) and database integration.

6.1 Core Database Systems Research

Storage: It is necessary to investigate the coexistence of two different storage mechanisms: row and column: currently two internal DBMSs seem the most efficient approach [12, 22]. More efficient row to column converters are needed to process deltas on streams and high velocity OLTP transactions. Hardware can be

exploited, especially RAM and SSDs. For instance, Hana [12] can efficiently query an entire database in RAM (say hundreds of GBs). So a given a query workload a carefully chosen subset of the database could be processed completely in RAM, like Shark [35]. Compressed blocks can be more efficiently read and written on SSDs. Support for diverse and complex data types is not well understood yet: fixed length storage helps address computation, whereas compression leads to variable length storage. Text data with a large number of keywords across a document collection represents a sparse data set with many opportunities for optimization. BLOBs are used to enable arrays and exploiting cache memory [18], but it is unclear if they can be combined with RLE compression. UDTs mixing simple data types and text need careful storage layout optimization.

Updating database: Processing transactions is a bad fit for columnar DBMSs, but they generally incorporate an internal row store in RAM to convert rows to columns. Timestamping has proven more effective for fast multicore CPUs [18, 22], leaving locking for legacy DBMSs. Given the speed of columnar DBMSs and their storage flexibility it may be better to do ELT, instead of ETL to integrate databases. Another potential improvement, enabled by new hardware, is to provide transaction processing and ad-hoc querying on the same DBMS when the database fits in RAM.

Indexing: Since ordering by every projection is infeasible some form of indexing is needed to complement ordering. Sparse B-trees are a solution [31], but it is unclear if they can benefit more general query processing. Hashing is used in main memory processing [22], but not on disk. Adaptive indexing can accelerate query processing as the workload varies, like database cracking.

Query processing: Row and column stores are competing and influencing each other. So it is necessary to investigate guidelines to add row DBMS features to columnar DBMSs and vice-versa. Bushy plans [22] versus shallow/simpler query trees [18]. We need new algorithms for automated physical design to compete with NoSQL systems. Fast long writes and faster seeks on SSDs can accelerate scans and even hash-based joins. Streams need revisiting joins (band joins, stream+table lookup join), and complex aggregations (especially approximate histograms). We need to revisit well-known analytic query problems with hybrid row and columnar processing: cubes (iceberg queries [13], sparse cube storage [29], horizontal aggregations [28]), recursive queries (transitive closure [4, 26]), skylines [16], and spatio-temporal data [13]. For ever-growing data sets approximation and sampling are needed (sample for plan cost estimation [18], view materialization, stream analysis over a window [37], dynamically reindexing continuously changing data [19]), but sampling requires materializing rows. Development of new database languages is needed: going beyond SQL, XML, Datalog, although adoption is a practical problem.

Parallel processing: Parallel merge/hash joins and GROUP BY aggregations remain challenging. New data partitioning and alignment algorithms are needed to improve data balancing. It is necessary to study the tradeoffs between traditional partitioning versus sharding [30]. Query workload optimization will be-

come more important: query scheduling, like Hadoop/MR job scheduling [9]. Fault tolerance with massive parallelism for slow queries is needed [22].

6.2 Big Data Management and Machine Learning

Storage and Querying: The main characteristics of Big Data are the three "Vs": Volume, Velocity and Variety [7, 14, 30], although this is being expanded with two additional Vs: veracity (many data sources, possibly with conflicting information) and value (usefulness). Volume is now represented by two data worlds: data warehouses and search engines. Variety is the hardest challenge [7]: data management issues are compounded by text (documents, files, web data, logs), matrices (vectors, arrays) spatio-temporal data (location, historical) and graphs (describing general relationships). Velocity is represented by streams: sensors and user log data. Currently, columnar DBMSs are very fast for wide denormalized tables [1, 22], slightly slower than row DBMSs for large narrow tables (i.e. normalized tables) [1], reasonably fast for data coming periodically in batches (data warehousing), OK for high velocity data (exploiting cache and multicore CPUs [37]), fair, but better than row DBMSs, for "variety" data with complex structure (graphs) or little or no structure (text, no schema). Since columnar DBMSs have more flexible storage than row DBMSs, they can be adapted to process text relaxing or automating the schema definition requirement using key-value pairs like Hadoop systems, thereby allowing storage of columns with varied text content. Further research is needed to query large graphs, especially stored by edge [26], analyzing tradeoffs between dense (e.g. quadratic number of edges, highly connected) and sparse graphs (e.g. linear number of edges, trees, disconnected). Columnar DBMSs show promise to store variable length text data with inconsistent information: innovative research is needed to automate schema definition on text data.

Machine learning: Considering previous research on row DBMSs [8, 17, 27] and popular Hadoop/MapReduce/Spark [8, 36], this is a categorization of problems in descending importance order: (1) Processing alternatives: is it better to build data mining tools working outside the DBMS realm? this approach affords flexibility and overcomes DBMS limitations to develop fast algorithms, but it leads to data management problems and I/O bottlenecks to transfer data [17, 27]. Therefore, it is necessary to determine a complexity/cost boundary of problems that cannot be computed efficiently even inside a columnar DBMS. (2) Scalable computation of statistical models with large data sets (hard) and high dimensionality (harder). The three major solutions are: internal integration with DBMS source code via cursors UDFs, SQL queries.Given their speed, columnar DBMSs look promising to compute models and graphs entirely with SQL queries, followed by UDFs exploiting denormalization and fast joins/aggregations. However, most algorithms require all columns, not a projection. Reducing the number of passes over the data set and data summarization seem orthogonal to storage by row or column: tradeoffs between both storage mechanisms need study. On the other hand, sampling requires row materialization. (3) Pattern discovery: association rules, sequences, Such algorithms need traversing the dimensions lattice,

which generally requires pointers, but may be faster with columns than rows. (4) Supporting broader mathematical processing, enabled by arrays: graph mining, matrix operators, linear algebra and key numerical methods like gradient descent. The integration of mathematical packages and libraries (e.g. R, Matlab, BLAS/LAPACK) with the DBMS is difficult and time consuming. Is it hacking? No, we believe there are indeed research issues: There is a programming language impedance mismatch with SQL, efficient data transfer and conversion in RAM are needed, memory and disk management is difficult when running on the same hardware. (5) Text and web data analytics are a better fit for Hadoop (MapReduce and Spark), but the following tasks need further study in modern DBMSs when documents and databases are inter-related: text preprocessing, ranking, ontologies, document classification and text summarization.

Acknowledgments

The first author thanks the guidance from Michael Stonebraker to understand query processing based on columnar storage, arrays of unlimited size to support mathematical analytics and lock-free transaction processing in main memory.

References

1. D.J. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *Proc. ACM SIGMOD Conference*, pages 967–980, 2008.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases : The Logical Level*. Pearson Education POD, facsimile edition, 1994.
3. A. Abouzied, K. Bajda-Pawlikowski, J. Huang, D.J. Abadi, and A. Silberschatz. HadoopDB in action: building real world applications. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 1111–1114. ACM, 2010.
4. F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. ACM SIGMOD Conference*, pages 16–52, 1986.
5. Peter Baumann, Alex Mircea Dumitru, and Vlad Merticariu. The array database that is not a database: file based array query answering in rasdaman. In *Advances in Spatial and Temporal Databases*, pages 478–483. Springer, 2013.
6. Ladjel Bellatreche, Soumia Benkrid, Ahmad Ghazal, Alain Crolotte, and Alfredo Cuzzocrea. Verification of partitioning and allocation techniques on teradata DBMS. In *Algorithms and Architectures for Parallel Processing - 11th International Conference, ICA3PP, Melbourne, Australia, October 24-26, 2011, Proceedings, Part I*, pages 158–169, 2011.
7. S. Ceri, E. Valle, D. Pedreschi, and R. Trasarti. Mega-modeling for big data analytics. In *Proc. ER Conference*, pages 1–15, 2012.
8. J. Cohen, B. Dolan, M. Dunlap, J. Hellerstein, and C. Welton. MAD skills: New analysis practices for big data. In *Proc. VLDB Conference*, pages 1481–1492, 2009.
9. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
10. D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

11. J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vost. *Numerical Linear Algebra for High-Performance Computers*. SIAM, 1998.
12. F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database: An architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.
13. H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2nd edition, 2008.
14. A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H. Jacobsen. Bigbench: towards an industry standard benchmark for big data analytics. In *Proc. ACM SIGMOD Conference*, pages 1197–1208. ACM, 2013.
15. A. Hameurlain and F. Morvan. Parallel relational database systems: Why, how and beyond. In *Proc. DEXA Conference*, pages 302–312, 1996.
16. J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2006.
17. J. Hellerstein, C. Re, F. Schoppmann, D.Z. Wang, E. Fratkin, A. Gorajek, K.S. Ng, and C. Welton. The MADlib analytics library or MAD skills, the SQL. *Proc. of VLDB*, 5(12):1700–1711, 2012.
18. S. Idreos, F. Groffen, N. Nes, S. Manegold, K.S. Mullender, and M.L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.
19. S. Idreos, M.L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column stores. In *Proc. ACM SIGMOD Conference*, pages 297–308, 2009.
20. A. Jacobs. The pathologies of big data. *Commun. ACM*, 52(8):36–44, 2009.
21. D. Jemal, R. Faiz, A. Boukorca, and L. Bellatreche. MapReduce-DBMS: An integration model for big data management and optimization. In *Proc. DEXA Conference*, pages 430–439, 2015.
22. A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear. The Vertica analytic database: C-store 7 years later. *PVLDB*, 5(12):1790–1801, 2012.
23. P.A. Larson, E.N. Hanson, and S.L. Price. Columnar storage in SQL Server 2012. *IEEE Data Eng. Bull.*, 35(1):15–20, 2012.
24. R. MacNicol and B. French. Sybase IQ multiplex - designed for analytics. In *Proc. VLDB Conference*, pages 1227–1230, 2004.
25. S. Manegold, P.A. Boncz, and M.L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(4):709–730, 2002.
26. C. Ordonez. Optimization of linear recursive queries in SQL. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(2):264–277, 2010.
27. C. Ordonez. Statistical model computation with UDFs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(12):1752–1765, 2010.
28. C. Ordonez and Z. Chen. Horizontal aggregations in SQL to prepare data sets for data mining analysis. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(4):678–691, 2012.
29. Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: shrinking the petacube. In *ACM SIGMOD Conference*, pages 464–475, 2002.
30. M. Stonebraker, D. Abadi, D.J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.
31. M. Stonebraker, D.J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E.J. O'Neil, P.E. O'Neil, A. Rasin, N. Tran, and S.B.

Zdonik. C-Store: A column-oriented DBMS. In *Proc. VLDB Conference*, pages 553–564, 2005.
32. M. Stonebraker, P. Brown, D. Zhang, and J. Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science and Engineering*, 15(3):54–62, 2013.
33. M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.
34. N. Tran, S. Bodagala, and J. Dave. Designing query optimizers for big data problems of the future. *PVLDB*, 11(6), 2013.
35. R.S. Xin, J.R., M. Zaharia, M.J. Franklin, S.S., and I. Stoica. Shark: SQL and rich analytics at scale. In *Proc. ACM SIGMOD Conference*, pages 13–24, 2013.
36. M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud USENIX Workshop*, 2010.
37. M. Zukowski and P. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012.