Survey of Large-Scale Data Management Systems for Big Data Applications

Lengdong Wu, Li-Yan Yuan, Jia-Huai You

Department of Computing Science
University of Alberta, Edmonton, Alberta, Canada, T6G 2E8.

E-mail: lengdong, yuan, you@cs.ualberta.ca

Abstract   Today, data is flowing into various organizations at an unprecedented scale. The ability to scale out for processing an enhanced workload has become an important factor for the proliferation and popularization of database systems. Big data applications demand and consequently lead to developments of diverse large-scale data management systems in different organizations, ranging from traditional database vendors to new emerging internet-based enterprises.

In this survey, we investigate, characterize and analyze the large-scale data management systems in depth and develop comprehensive taxonomies for various critical aspects covering the data model, the system architecture and the consistency model. We map the prevailing highly scalable data management systems to the proposed taxonomies, not only to classify the common techniques but also to provide a basis for analyzing current system scalability limitations. To overcome these limitations, we predicate and highlight the possible principles that future efforts need to be undertaken for the next generation large-scale data management systems.

Keywords   Data Model; System Architecture; Consistency Model; Scalability

1 Introduction

Data is flowing into organizations at an unprecedented scale in the world. Data volumes collected by many companies are doubled in less than a year or even sooner. The growing speed is faster than the "Moore's Law", which predicts that the general-purpose hardware and software solutions that advance at the rate of Moore's Law will not be able to keep pace with the exploding data scale [1]. The pursuit for tackling the challenges posed by the big data trend has given rise to a plethora of data management systems characterized by high scalability. Diverse systems for processing big data explore various possibilities in the infrastructure design space. A notable phenomenon is the NoSQL (Not Only SQL) movement that began in early 2009 and is evolving rapidly.

In this work, we provide a comprehensive study of the state-of-the-art large-scale data management systems for the big data application, and also conduct an in-depth analysis on the critical aspects in the design of different infrastructures. We propose taxonomies to classify techniques based on multiple dimensions, in which every high scalable system is able to find its position. A thorough understanding of current systems and a precise classification are essential for analyzing the scalability limitations and ensuring a successful system transi-tion from enterprise infrastructure to the next generation of large-scale infrastructure.

1.1 Systems for Big Data Application

To date, the trend of "Big Data" is usually characterized by the following well-known cliché:

- *Volume*: Excessive data volumes and a large number of concurrent users require substantially throughput raising for the systems.

- *Velocity*: Data is flowing in at an unprecedented speed and needs to be dealt with in a timely manner.

- *Variety*: Data comes in all types of formats, from the structured relational data to the unstructured data.

- *Veracity*: Inconsistency or uncertainty of data, due to the quality of the data source or transmission latency, will jeopardize the utility and integrity of the data.

The "big data" trend has imposed challenges on the conventional design and implementation of data management systems. In particular, the ability to scale out for processing an enhanced workload has become an important factor for the proliferation and popularization of data management systems.

When considering the spectrum of data management systems, we first have traditional relational database systems (RDBMSs) that provide low latency and high throughput of transaction processing, but lack the capacity of scale-out. As expected, traditional database vendors have recently developed their own system appliances in response to the high scalability requirement. Typically, Oracle Exadata [1], IBM Netezza [2] and Teradata [3] exploit the declarative nature of relational query languages and deliver high performance by leveraging a massively parallel fashion within a collection of storage cells. Oracle Exalytics [2] is the industrial pioneer to use terabytes of DRAM, distributed across multiple processors, with a high-speed processor interconnect architecture that is designed to provide a single hop access to all the memory.

Some recently proposed "New SQL" relational databases (relative to NoSQL) aim to achieve the scalability same as NoSQL while preserving the complex functionality of relational databases. Azure [3] is a parallel run-time system, utilizing specific cluster control with minimal invasion into the SQL Server code base. Some research prototypes, such as Rubato DB [4], H-store [5], later commercialized into VoltDB[4], and C-Store [6], the predecessor of Vertica[5], also provide their tentative solutions for the NewSQL implementation.

The need of highly available and scalable distributed key-value data stores with reliable and "always-writable" properties, leads to the development of Amazon Dynamo [7] and Yahoo! PNUTS [8]. An open-source clone of Dynamo, Cassandra [9], has also been developed by the Apache community. Oracle NoSQL Database [10] using Oracle Berkeley DB as the underlying data storage engine provides flexible durability and consistency polices. Similar systems such as Voldemort[6], SimpleDB[7], etc., are all categorized as key-value data stores. Key-value data stores are characterized as simplified highly scalable databases addressing properties of being schema-free, simple-API, horizontal scalability and relaxed consistency.

---

[1]Technique overview of the Oracle Exadata Database Machine and Exadata Storage Server. Oracle White Paper, 2012.

[2]The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. IBM Red Paper, 2011.

[3]Teradata past, present, and future. `http://isg.ics.uci.edu/scalable_dml_lectures2009-10.html`

[4]https://voltdb.com/

[5]http://www. vertica.com/

[6]http://project-voldemort.com/

[7]http: //aws.amazon.com/en/simpledb/

Google responds to the web-scale storage challenges by developing a family of systems. Google File System (GFS) [11] is a distributed file system for large distributed data-intensive applications, providing with an OS-level byte stream abstraction on a large collection of commodity hardware. The BigTable [12] is a hybrid data storage model built on GFS. Megastore [13] and Spanner [14] are two systems over the BigTable layer. Megastore blends the scalability and the fault tolerance ability of BigTable with transactional semantics over distant data partitions. Spanner is a multi-versioned, globally-distributed and synchronously replicated database by adopting True Time, which combines an atomic clock with a GPS clock for synchronization across world-wide datacenters. HBase[8] and Hypertable[9] provide open-source versions of Google's BigTable.

Google developed the MapReduce framework that is highly scalable and parallel for big data processing [15]. Taking the released MapReduce paper as the guideline, open-source equivalents were developed as well, such as the Apache Hadoop MapReduce platform built on the Hadoop Distributed File System

(HDFS) [16]. Numerous NoSQL systems based on the MapReduce and Hadoop utilize a large collection of commodity servers to provide high scalability. For example, a set of systems with high-level declarative languages, including Yahoo! Pig [17], Microsoft SCOPE [18] and Facebook Hive [19], are realized to compile queries into the MapReduce framework before the execution on the Hadoop platform. Greenplum [20] integrates the ability to write MapReduce functions over data stored in their parallel database products. HadoopDB [21] is to connect multiple single-node database systems using Hadoop as the task coordinator and the network communication layer. Queries are parallelized across nodes using the MapReduce framework. There are also a number of commercialized systems that combine parallel DBMSs with the MapReduce framework such as Cloudera[10], Teradata Aster[11], etc.

The landscape of representative data management systems can be divided into two categories, as summerized in Figure 1. Relational zone includes systems supporting relational data, such as conventional relational databases, appliances and New SQL systems. Systems in the non-relational zone evolve from

---

[8]http://hbase.apache.org/

[9]http://hypertable.org/

[10]https://www.cloudera.com

[11]http://www.asterdata.com/

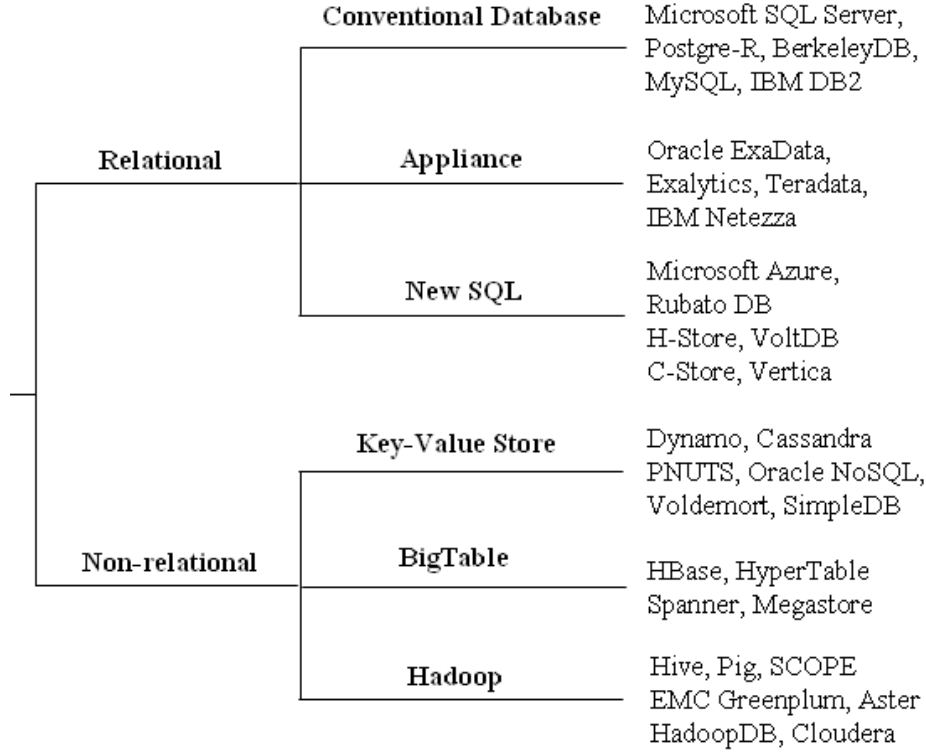| Conventional Database | Microsoft SQL Server, Postgre-R, BerkeleyDB, MySQL, IBM DB2 |

Fig. 1. Landscape of Representative Data Management Systems

emerging infrastructures such as the key-value store, BigTable and Hadoop.

1.2 Synopsis

Having set the stage for large-scale data management systems, in this survey, we delve deeper to present our insights into the critical aspects of big data applications. We study the essential aspects as follows:

- **Data model**. We capture both physical and conceptual aspects of the data model for large-scale data management systems.

- **System architecture**. We give a comprehensive description of diverse architectures by examining and distinguishing the way how various modules are orchestrated within a system and how the data flow control logic is distributed throughout the system.

- **Consistency model**. We investigate progressive consistency levels applied by existing systems, and analyze various consistency models ranging from the weakest one to the strongest one.

Based on our taxonomies and analysis, we then identify the principles for the implementation of large scale data management systems including:

- Facilitating the close integration of data modeling and data partitioning by the **hybrid storage layout**;

- Obtaining high scalability with the **SEDA/MapReduce architecture**;

- Scaling out the concurrency control protocol based on **timestamps**;

- Developing **restrictions** on strong consistency models properly for scalability;

- Overcoming the weakness of BASE through stronger consistency models such as **BASIC**.

The remainder of this survey is organized as follows. Section 2 discusses the different designs of data model from two aspects of the physical layout and the conceptual schema. Section 3 undertakes a deep study on the diverse scalable architecture designs. Section 4 presents the different levels of consistency models, and the trade-off for scalability. Section 5 provides our analysis based on the taxonomies, and proposes priciples on the implementation for large-scale data management systems. Section 6 concludes this survey.

2 Data Model

Data model consists of two essential levels: the physical level and the conceptual level. The details of how data is stored in the database belong to the physical level of data modeling[12]. The schemas specifying the structure of the data stored in the database are described in the conceptual level.

2.1 Physical Level

A key factor affecting the performance of any data management system is its storage layout on the physical layer used to organize the data on database hard disks. There are three mechanisms to map the two-dimensional tables onto the one-dimensional physical storage, i.e., *row-oriented layout*, *column-oriented layout*, and *hybrid-oriented layout*.

Row-oriented Layout. Data has been organized within a block in a traditional row-by-row format, where all attributes data of a particular row is stored sequentially within a single database block. Traditional DBMSs towards ad-hoc querying of data tend to choose the row-oriented layout.

Column-oriented Layout. Data is organized in a significant deviation of the row-

---

[12]By physical level we mean a lower level of storage schemas, not actual file structures on disk.

oriented layout. Every column is stored separately in the column-oriented layout and values in a column are stored contiguously. Analytic applications, in which attribute-level access rather than tuple-level access is the frequent pattern, tend to adopt the column-oriented layout. They can then take advantage of the continuity of values in a column such that only necessary columns related with the queries are required to be loaded, reducing the I/O cost significantly [22].

Hybrid-oriented Layout. The design space of the physical layout is not limited to merely row-oriented and column-oriented layouts, but rather that there is a spectrum between these two extremes, and it is possible to build the hybrid layout combining the advantages of purely row and column oriented layouts.

Hybrid-oriented layout schemas are designed based on different granularities. The most coarse-grained granularity essentially adopts different layouts on different replicas like fractured mirrors [23]. The basic idea is straightforward: rather than two disks in a mirror being physically identical, they are logically identical in which one replica is stored in the row-oriented layout while the other one is in the column-oriented layout. Fractured mirror can be regarded as a new form of RAID-1, and

the query optimizer decides which replica is the best choice for corresponding query execution. The fine-grained hybrid schema [24, 25] integrates row and column layouts in the granularity of individual tables. Some parts of the table are stored with the row-oriented layout, while other parts apply the column-oriented layout. An even finer schema is based on the granularity of disk blocks. Data in some blocks is aligned by rows while some is aligned by columns. To some extend, we can consider that, *row-oriented layout, column-oriented layout* are special extreme cases of *hybrid-oriented layout.*

2.2   Conceptual Level

Obtaining maximum performance requires a close integration between the physical layout and the conceptual schema. Based on the interpretation of data, three different conceptual data structures can be defined, i.e., *unstructured data store*, *semi-structured data store* and *structured data store.*

Unstructured data store. All data items are uninterrupted, isolated and stored as a binary object or a plain file without any structural information. Unstructured data store takes the simplest data model: a map allowing requests to put and retrieve values per key.

Extra efforts are required from programmers for the interpretation on the data. Under the restricted and simplified primitives, the key-value paradigm favors high scalability and performance advantages [7, 8, 11]. Due to the lack of structural information to extract data items separately, the row-oriented physical layout is the only choice for the unstructured data store.

Semi-structured data store. A semi-structured data store is used to store a collection of objects that is richer than the uninterrupted, isolated key/value pairs in the unstructured data store. A semi-structured data store, being schemaless, has certain inner structures known to applications and the database itself, and therefore can provide some simple query-by-value capability, but the application-based query logic may be complex [12]. Because of its nature of schemaless, a semi-structured data store can only adopt row-oriented layout on the physical layer.

Structured data store. A structured data store is used to store highly structured entities with strict relationships among them. Naturally, a structured data store is defined by its data schema, and usually supports comprehensive query facilities. As a representative of

structured data store, the relational database organizes data into a set of tables, enforces a group of integrity constraints, and supports SQL as the query language.

2.3 Data Model Taxonomy

Based on the classification of the physical layout and the conceptual schema, we analyze currently prevailing database systems and categorize them in an appropriate taxonomy, as demonstrated in Figure 2, based on our observations outlined below.

Amazon's Dynamo [7], Yahoo! PNUTS [8], Voldemort and SimpleDB are the typical systems belonging to the category with the row-oriented physical layout and the unstructured conceptual data store, since they are built on the simple key-value paradigm by storing data as binary objects (i.e., blobs) identified by unique keys. These systems are all unstructured data stores that can only use the row-oriented physical layout.

Google's Bigtable [12] together with Cassandra [9], HBase, HyperTable, which are Bigtable-like systems, are representatives of semi-structured data stores built on the hybrid-oriented physical layout. They treat each individual table as a sparse, distributed, multi-dimensional sorted map that provides the semi-
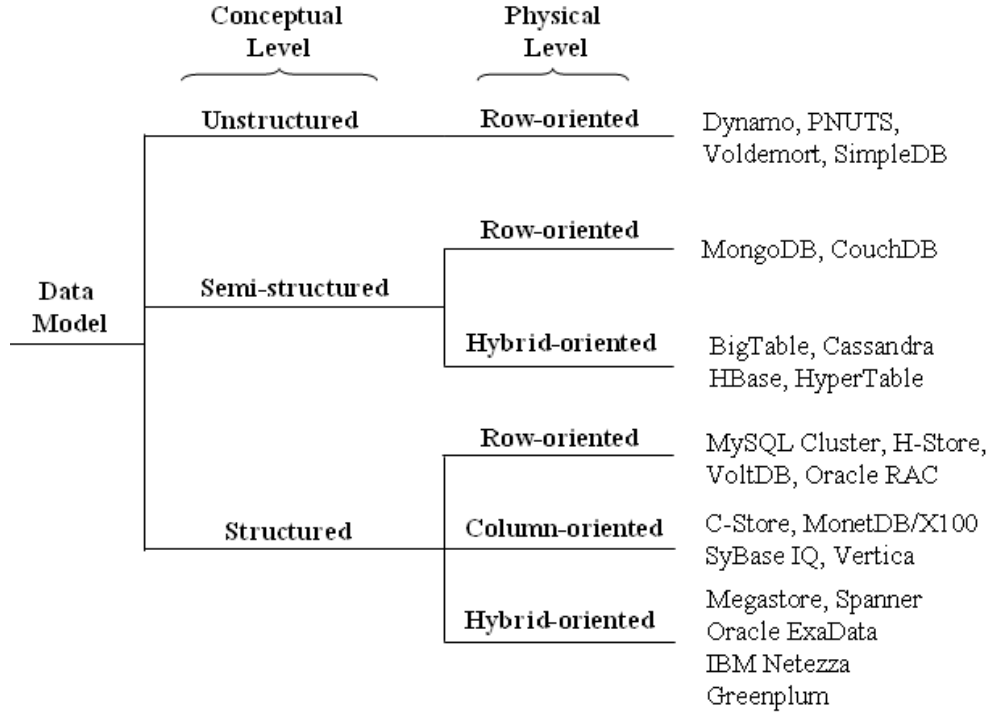
---

[13]http://couchdb.apache.org/

[14]http://www.mongodb.org/

Fig. 2. Taxonomy of Data Model

structured data.

CouchDB[13] and MongoDB[14] considered as document stores[15], are another typical class of semi-structured data stores while using the row-oriented physical layout. Data in a document store is serialized from XML or JSON formats so that row-oriented layout is applied, similar to key-value stores.

C-store [6, 26] supports the relational structured data model, whereas tables are stored column-oriented physically. MonetDB/X100 [27, 28] and commercial systems Sybase IQ[16] and Vertica adopt the similar idea of C-store. These systems benefit greatly from data caching and compressing techniques. Having data from each column with the same data type and low information entropy stored close together, the compression ratio can be dramatically enhanced to save a large amount of storage.

Megastore [13] and Spanner [14] define a structured data model based on relational tables stored on Bigtable. Since they are built on top of Bigtable, the hybrid layout is applied on the physical level. Same as traditional relational databases, the data model is declared in a schema. Tables are either entity group root tables or child tables, which must declare a sin-

---

[15]http://en.wikipedia.org/wiki/Document-oriented_database

[16]http://sybase.com/

gle distinguished foreign key referencing a root table.

Oracle's Exadata , IBM's Netezza Server and Greenplum [20] evolved from traditional parallel database systems, and thus support the structured data store. Furthermore, Exadata introduces Hybrid Columnar Compression (HCC) in the granularity of disk blocks. HCC employs the similar idea of Partition Attributes Across (PAX) [29] combined with compression. Netezza integrates row and column oriented layouts on each individual table. Greenplum provides multiple storage mechanisms with a variety of formats for different level of compression modes. The column-oriented store with the slightly compressed format is applied for data that is updated frequently, and append-only tables are using the row-oriented store with the heavily compressed format. These systems adopt the hybrid-oriented layout.

3   System Architecture

The system architecture is the set of specifications and techniques that dictate the way how various modules are orchestrated within a system and how data processing logic works throughout the system.

In this section, we are going to classify systems according to diverse architectures. There are four historical shifts in the architecture technology behind large-scale data management systems:

1. The invention of databases on cluster of processors (single or multi-core) with shared memory.

2. The improvement of databases on cluster of processors with distributed memory but common storage disks.

3. The rise of parallel databases processing on shared-nothing infrastructure.

4. The popularization of the MapReduce parallel framework and the distributed file system.

3.1   SMP on Shared-Memory Architecture

The symmetric multi-processing (SMP) on shared-memory architecture, as illustrated in Figure 3, involves a pool of tightly coupled homogeneous processors running separate programs and working on different data with sharing common resources such as memory, I/O device, interrupt system and system bus. The single coherent memory pool is useful for sharing data and communication among tasks. This architecture is fairly common that most conventional database management systems have been deployed on such high-end SMP architectures.

However, a small-scale SMP system consisting of a few processors is not capable of managing large-scale big data processing. It can be scaled "up" by adding additional processors, memories and disks devices, but is inevitably bounded by the resources limitation. In particular, when data volumes are increasing enormously, the memory bus bandwidth will be the ceiling for scaling-up, and similarly I/O bus bandwidth can also be clogged.

In addition, the initial expense of scaling up SMP server is quite high due to the larger capabilities and often more complex architectures [30]. It has been observed that the efficiency, scalability, and cost effectiveness of SMP systems degrade beyond 32 modern high performance microprocessors [17]. The SMP on shared-memory architecture has the disadvantage of limited scalability.

3.2 MPP on Shared-Disk Architecture

The massively parallel processing (MPP) on shared-disk architecture is built on top of SMP clusters executing in parallel while sharing a common disk storage, as demonstrated in Figure 4. Each processor within an SMP cluster node shares the memory with its neighbors and accesses to the common storage across a shared I/O bus.

The shared-disk infrastructure necessitates disk arrays in the form of a storage area network (SAN) or a network-attached storage (NAS) [31]. For instance, Oracle and HP grid solution orchestrates multiple small server nodes and storage subsystems into one virtual machine based on the SAN [30]. Unlike the shared-memory infrastructure, there is no common memory location to coordinate the sharing of the data. Hence explicit coordination protocols such as cache coherency [32] and cache fusion [33] are needed [30].

The MPP on shared-disk architecture is commonly used in several well-known scalable database solutions. Two notable systems are Oracle Exadata Database Machine and IBM Netezza Performance Server (NPS) . Exadata is a complete, pre-configured Oracle system that combines Oracle RAC[18] with new Exadata Storage Servers. Exadata improves parallel I/O and filters only data of interest before transmitting. This process of filtering out extraneous data as early in the data stream as possible close to the data source can minimize the I/O bandwidth bottleneck and free up downstream components such as CPU and memory, thus having a significant multiplier effect on the performance. Netezza integrates server, storage and database all in a single com-

---

[17]Scaling-Up or Out. IBM Performance Technical Report. 2002.

[18]http://www.oracle.com/technetwork/products/clustering/overview/index.html

pact platform. It proposes Asymmetric Massively Parallel Processing (AMPP) mechanism with Query Streaming technology that is an optimization on the hardware level .
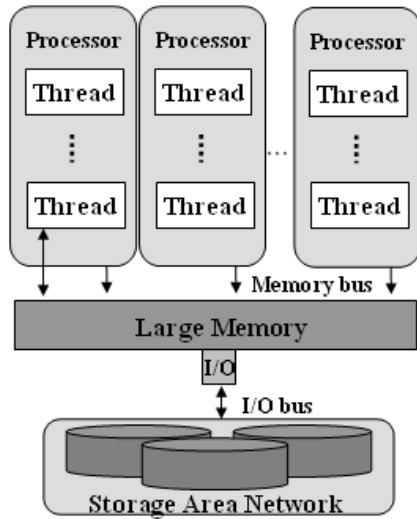

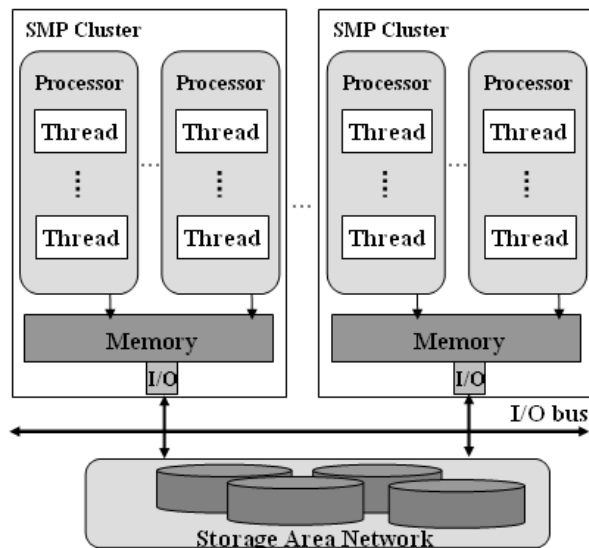Fig. 3. SMP on Shared-Memory Architecture
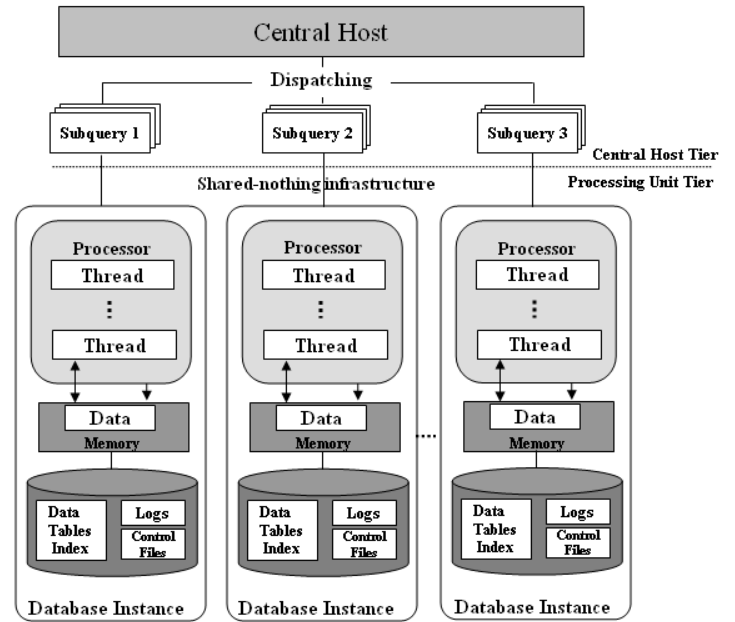

Fig. 4. MPP on Shared-Disk Architecture

Fig. 5. Sharding on Shared-Nothing Architecture

3.3 Sharding on Shared-nothing Architecture

Based on the idea that data management systems can be parallelized to leverage multiple commodity servers in a network to deliver increased scalability and performance, the parallelism on the shared-nothing infrastructure was coined for the new computing clusters. The sharding on the shared-nothing architecture is currently widely used in large-scale data management systems [3, 7, 8, 9, 11, 12, 13].

In order to harness the power of this architecture, data is partitioned across multiple computation nodes. Each node hosts its own independent instance of the database system with operating on its portion of data. Each

node is highly autonomous, performing its own scheduling, storage management, transaction management and replication. The autonomy allows additional nodes to be involved without concerning about interruption with others.

Sharding on shared-nothing architecture has a two-tier system design, as shown in Figure 5. The lower processing unit tier is composed of dozens to hundreds of processing machines operating in parallel. All query processing is decoupled at the processing unit tier. In the host tier, the assigned coordinator receives queries from clients and divides the query into a sequence of sub-queries that can be executed in parallel, and dispatches them to different processing units for execution. When processing units finish, the central host collects all intermediate results, handles post-processing and delivers results back to the clients. There are two flavors of this architecture that are *centralized topology* and *decentralized topology*.

Centralized Topology. Centralized topology utilizes a dedicated centralized coordinator to manage the system-wide membership state. The central server hosts the entire metadata and periodically communicates with each data server via heartbeat messages to collect the status of each member. The central server also takes charge of activities, typically including

identifying the nodes that own the data with the key, routing the request to the nodes and integrating for the responses. The centralized topology simplifies the design and implementation of the complex architecture since the central node has an authoritative view of the whole system [8].

To prevent the central master server from easily becoming the bottleneck of the heavy workload, shadow master mechanism is employed [11]. The key idea is to separate the control flow and the data flow of the system. The central master is only responsible for the metadata operation, while clients communicate directly with the data servers for reads and writes bypassing the central master. This design also delivers high aggregate throughput for high concurrent readers and writers performing a variety of tasks.

Decentralized Topology. Unlike the centralized topology, systems such as Dynamo [7] and Cassandra [9] choose implementation of decentralized topology. All nodes take equal responsibility, and there are no distinguished nodes having special roles. This decentralized peer-to-peer topology excels the centralized one on the aspect of single point failure and workload balance. The gossip-based membership protocol [34] is a classical mechanism to ensure

that every node keeps a routing table locally and is aware of the up-to-date state of other nodes. Consistent hashing [35] is widely used in the decentralized topology implementation. Consistent hashing is a structure for looking up a server in a distributed system while being able to handle server failures with minimal effort. A client can send requests to any random node, and the node will forward the requests to the proper node along the ring.

3.4 MapReduce/Staged Event Driven Architecture

In the last decade, the importance of shared-nothing clusters was enhanced in the design of web services. Interesting architectures have been proposed to deal with massive concurrent requests on large data volumes of excessive user basis. One representative design is the well-known MapReduce framework for processing large data sets [15]. Another design is the Staged Event-Driven Architecture (SEDA), which is intended to allow services to be well-conditioned for loading, preventing resources from being over-committed when the demand exceeds service capacity [35].

Applications programmed with MapReduce framework are automatically parallelized and executed on a large cluster of commodity servers. The framework consists of two abstract functions, Map and Reduce, which can be considered as two different stages as well. The Map stage reads the input data and produces a collection of intermediate results; the following Reduce stage pulls the output from Map stage, and processes to final results. The trend of applying MapReduce framework to scale out configurations with lower-end, commodity servers has become popular, due to the drop in prices of the hardware and the improvement in performance and reliability.
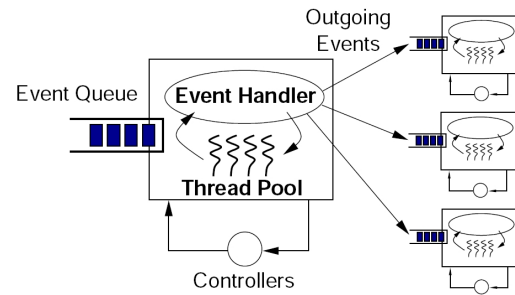
Fig. 6. Staged Event Driven Architecture

Staged Event-Driven Architecture (SEDA) is designed based on the event-driven approach that has been introduced and studied for various software applications, such as Dynamic Internet Servers and high performance DBMSs [35, 36]. The event-driven approach implements the processing of individual task as a finite state machine (FSM), where transitions between states in the FSM are triggered by events. The basic idea of this architecture is that a software system is constructed as a

network of staged modules connected with explicit queues, as illustrated in Figure 6 [35]. SEDA breaks the execution of applications into a series of stages connected by explicitly associated queues. Each stage represents a set of states from the FSM, and can be regarded as an independent, self-contained entity with its own incoming event queue. Stages pull a sequence of requests, one request at a time, off their incoming task queue, invoke the application-supplied event handler to process requests, and dispatch outgoing tasks by pushing them into the incoming queue of the next stage. Each stage is isolated from one another for the purpose of easy resource management, and queues between stages decouple the execution of stages by introducing explicit control boundaries [35]. It has been shown that the aforementioned MapReduce framework can also be regarded as an architecture based on SEDA, and the basic MapReduce framework resembles the two-staged SEDA architecture. The general MapReduce extensions [37, 38], introducing pipelined downstream data flow between multiple functional MapReduce pairs, behave identically as SEDA [49].

There has been some recent work on bringing ideas from MapReduce/SEDA to database management systems. The SEDA design has been applied to improve the staged database

performance through exploiting and optimizing locality at all levels of the memory hierarchy of the single symmetric multiprocessing system at the hardware level [36]. Some systems aim to integrate query construction into the MapReduce framework to allow greater data independence, automatic query optimization, and instruction reusability at the query level [17, 18]. There are also attempts to program the MapReduce/SEDA over high performance parallel databases as a hybrid solution at the systems level [19, 20, 21, 39].

3.5　System Architecture Taxonomy

Based on the above analysis, we present the taxonomy of the large-scale data management system architecture in Figure 7.

Due to the long-time popularity of the shared-memory multi-threads parallelism, almost all major traditional commercial DBMS providers support products with the SMP on shared-memory architecture, such as Microsoft SQL Server, Oracle Berkeley DB and Postgres-R, to name a few.

Microsoft Azure server [3] is built on Microsoft SQL Server and uses centralized topology over the shared-nothing infrastructure. This architectural approach is to inject the specific cluster control with minimal invasion into the MS SQL Server code base, which retains
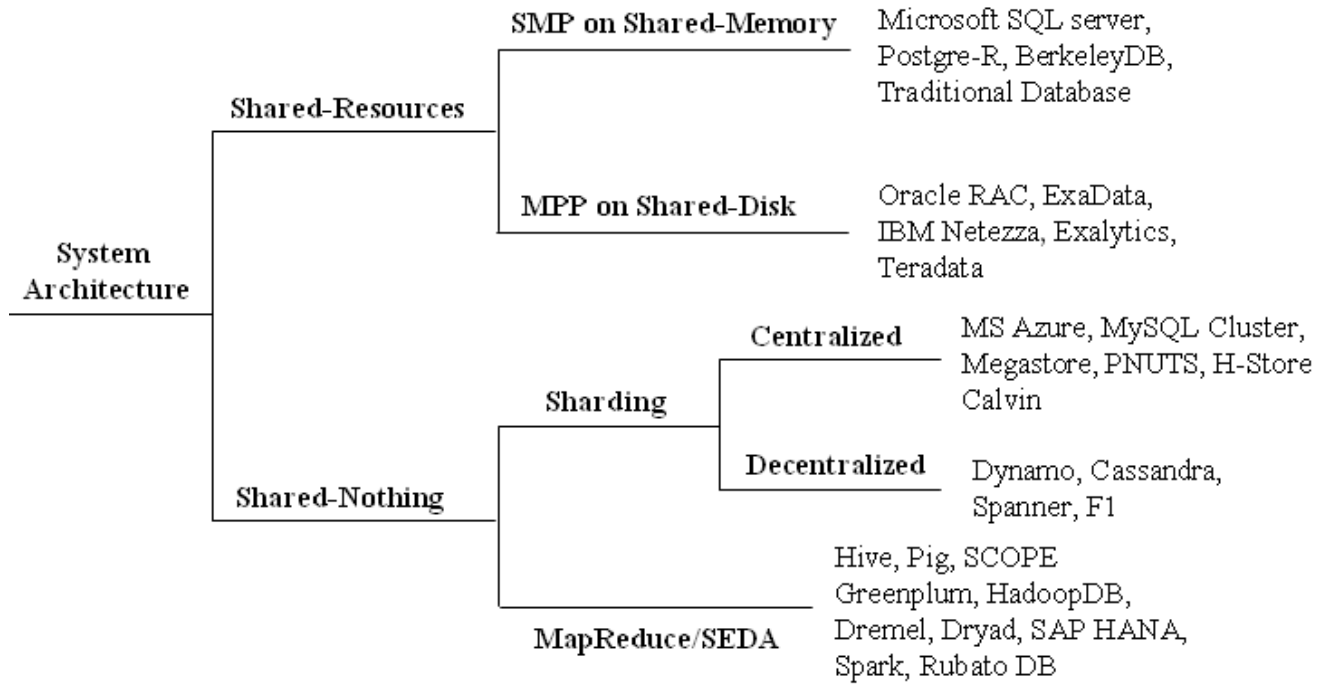
Fig. 7. Taxonomy of system architecture

much of the relational features of MS SQL Server. To enhance the scalability, MS Azure also assembles multiple logical databases to be hosted in a single physical node, which allows multiple local database instances to save on memory with the internal database structures in the server.

MySQL Cluster [19] applies a typical sharding on shared-nothing architecture based on MySQL. Data is stored and replicated on individual data nodes, where each data node executes on a separate server and maintains a copy of the data. MySQL Cluster automatically creates node groups from the number of replicas and data nodes specified by the user.

Each cluster also specifies the central management nodes.

H-Store [5] is a highly distributed relational database that runs on a cluster of main memory executor nodes on shared-nothing infrastructure. H-Store provides an administrator node within the cluster that takes a set of compiled stored procedures as inputs.

Megastore [13] is a higher layer over BigTable [12]. Megastore blends the scalability of Bigtable with the traditional relational database. Megastore partitions data into entity groups, providing full ACID semantics within groups, but only limiting consistency across them. Megastore relies on a highly available

---

[19]MySQL Cluster Architecture Overview. MySQL Technical White Paper, 2005.

and persistent distributed lock service for master election and location bootstrapping.

Yahoo! PNUTS [8] is a massively parallel and geographically distributed system. PNUTS uses a publish/subscribe mechanism where all updates are firstly forwarded to a dedicated master, and then the master propagates all writes asynchronously to the other data sites.

Calvin [40] is designed to serve as a scalable transactional layer above any storage system that implements a basic distributed non-transactional storage. Calvin organizes the partitioning of data across the storage systems on each node, and orchestrates all network communication that must occur between nodes in the course of transaction execution with optimized locking protocol.

System above all elect and utilize certain logically central nodes to manage the coordination of the whole cluster, thus they all belong to the centralized topology category. Dynamo [7], Cassandra [9] and Spanner [14] opt the symmetric structure on the decentralized topology over the centralized one based upon the understanding that symmetry in decentralization can simplify the system provisioning and maintenance. Systems with the decentralized topology basically employ a distributed agreement and group membership protocol to coordinate

actions between nodes in the cluster.

Dynamo uses techniques originating in the distributed system research of the past years such as DHTs [41], consistent hashing [42], quorum [43], etc. Dynamo is the first production of system to use the synthesis of all these techniques [7].

Facebook Cassandra [9] is a distributed storage system for managing very large amounts of structured data spread out across many commodity servers. Cassandra brings together the data model from the BigTable and the distributed system technologies from Dynamo.

Spanner [14] is a scalable, multi-version, globally-distributed database system based on the "True Time" API, which combines an atomic clock and a GPS clock to timestamp data so that it can then be synchronized across multiple machines without the need of centralized control. F1 [44] is built on top of Spanner, which provides extremely scalable data storage, synchronous replication, and strong consistency and ordering properties.

Hive [19, 39], Scope [18] and Pig latin [17], built on top of Hadoop, compile SQL-like declarative queries into a directed acyclic graph of MapReduce jobs executed on Hadoop. They systematically leverage technologies from both parallel databases and MapReduce framework

throughout the software stack.

Spark[45] introduces resilient distributed dataset that lets applications keep data in memory across queries, and automatically reconstruct data nodes in failure. Its parallel operations fit into the iterative MapReduce which extends the traditional framework to support iterative data analysis. Spark focuses on applications that reuse a set of data across multiple parallel operations. Shark [46] is a low-latency system built on Spark, which can efficiently combine SQL engine and machine learning workloads, while supporting fine-grained fault recovery.

Greenplum [20] is a hybrid system that enables to execute write functions in SQL queries across multiple nodes in MapReduce style. It makes the effort for parallel loading of Hadoop data, retrieving data with MapReduce, and accessing Hadoop data by SQL.

HadoopDB [21] is built based on the idea of providing Hadoop access to multiple single-node DBMS servers and pushing data as much as possible into the engine. HadoopDB is to connect multiple single-node database systems by using Hadoop as the task coordinator and the network communication layer. Queries are parallelized across nodes using the MapReduce framework.

Dremel [47] uses a multi-level serving tree to execute queries that resemble the SEDA during data process. Each query gets pushed down to the next level in the serving tree, and is rewritten at each level. The result of the query is assembled by aggregating the replies received from the leaf servers at the lowest level of the tree which scan the tablets in the storage layer in parallel.

Dryad [49] is based on a direct acyclic graph (DAG) that combines computational vertices with communication channels to form a data flow graph. The vertices of the graph are on a set of available computers, communicating through files, TCP pipes, and shared-memory FIFOs. Dryad schedules vertices to run simultaneously on multiple computers for parallelism. The arbitrary execution data flow through the communication channel in Dryad is identical to the SEDA.

SAP HANA [50] database is the core of SAP's new data management platform. It introduces the calculation graph model that follows the classical data flow graph principle. The calculation model defines a set of intrinsic operators based on different types of nodes. Source nodes represent persistent table structures or the outcome of other calculation graphs. Inner nodes reflect logical operators consuming one or multiple incoming data flows and produce any number of outgoing data

flows.

Rubato DB [4] is constructed as a network of grid encapsulated modules on the shared-nothing infrastructure. Each staged grid encapsulated module runs on a grid node and accesses only the data stored in the DAS (Direct-Attached Storage) of the node. All grid nodes are connected by a (high speed or otherwise) network. Rubato DB utilizes different communication channels among all stages, depending on locations of stages and/or the system resources. A set of software instructions is also introduced to specify all basic operations and/or data packets. Each instruction carries all necessary information required for a request or a data packet, and the system will process in a relay-style by passing the instruction from one staged grid module to the next one until it is completed.

4 Consistency Model

One of the challenges in the design and implementation of big data management systems is how to achieve high scalability without sacrificing consistency. The consistency property ensures the suitable order and dependency of operations throughout the system, helping to simplify application development. However, most large-scale data management systems currently implement a trade-off between scalability and consistency, in that strong consistency guarantees, such as ACID [48], are often renounced in favor of weaker ones, such as BASE [8]. In this section, we are going to classify systems according to different consistency levels based on ACID and BASE.

4.1 ACID Properties

There are a set of properties that guarantee that database transactions are processed reliably, referred to as ACID (*Atomicity*, *Consistency*, *Isolation*, *Durability*) [48]. Database management systems with ACID properties provide different isolation levels, mainly include *serializability*, *snapshot isolation*, and *read committed* [51].

Serializability, the highest isolation level, guarantees that the concurrent execution of a set of transactions results in a system state that would be obtained if transactions were executed serially, i.e., one after the other. It is typically implemented by pessimistic reads and pessimistic writes, achieving the condition that unless the data is already updated to the latest state, the access to it is blocked.

Snapshot isolation is a multi-version concurrency control model based on optimistic reads and writes. All reads in a transaction can see a consistent committed snapshot of the database. A data snapshot is taken when the

snapshot transaction starts, and remains consistent for the duration of the transaction. Restrictions such as "The First-Committer Wins" rule allow snapshot isolation to avoid the common type of lost update anomaly [52].

Read committed, allowing applications trading off consistency for a potential gain in performance, guarantees that reads only see data committed and never see uncommitted data of concurrent transactions.

If we use the > symbol to represent the stronger relationship among two isolation levels, it is shown that [51]:

$$serializability > snapshot\ isolation$$
$$> read\ committed$$

To provide high availability and read scalability, *synchronous replication* is an important mechanism. With *synchronous replication*, rather than dealing with the inconsistency of the replicas, the data is made unavailable until update operations are propagated and completed in all or most of replicas. Update operations may be rejected and rolled back if they fail to reach all or a majority of the destination replicas within a given time. When *serializable* consistency is combined with *synchronous replication*, we can achieve *one-copy serializability* [52], in which the execution of a set of transactions is equivalent to executing the transactions in the serial order within only one single up-to-date copy. Similarly, combining *read committed* and *snapshot isolation* with *synchronous replication*, *one-copy read committed* and *one-copy snapshot isolation* can be obtained, respectively [53].

4.2 BASE Properties

The ACID properties work fine with horizontally scalable, relational database clusters. However, they may not well fit in the new unstructured or non-relational, large-scale distributed systems, in which flexible key/value paradigm is favored and the network partition or node failure can be normal rather than rare. Naturally, many large-scale distributed key-value store systems, such as Amazon Dynamo [7], Yahoo! PNUTS [8] and Facebook Cassandra [9], choose BASE, a consistency model weaker than ACID. The BASE, standing for *Basically Available, Soft-state, Eventually consistent*, can be summarized as: the system responses basically all the time (Basically Available), is not necessary to be consistent all the time (Soft-state), but has to come to a consistent state eventually (Eventual consistency) [54].

Various BASE consistency models have been specified, and thus we first categorize these models and present multiple system im-

plementations to demonstrate different levels of consistency model.

4.2.1 Eventual Consistency

Eventual consistency, one of the fundamental requirements of BASE, informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that data item will return the last updated value.

Even though a system with eventual consistency guarantees to "eventually" converge to a consistency state, it increases the complexity of distributed software applications because the eventual consistency does not make safety guarantees: an eventually consistent system can return any value before it converges [54, 55]. Eventual consistency may not provide a single image system since it makes no promise about the time intervals before the convergence is reached. In addition, the value that eventually achieved is not specified. Thus, additional restriction is required from applications to reason the convergence [56, 57, 58].

4.2.2 Causal Consistency

Causality is described as an abstract condition that ensures execution in a cluster agrees on the relative ordering of operations which are causally related. Conditions of causality based on reads and writes derive from causal memory

[59]. Causal consistency guarantees the relative ordering of read and write operations that are causally related [55, 60]. Causality is described as an abstract condition that ensures execution in a cluster agrees on the relative ordering of causally related operations. The causality exists between two operations op_1 and op_2, if one of the following conditions holds:

(a) *write-read* dependency: $op_1 \rightarrow op_2$, if op_1 is a write operation that writes the data item x, and op_2 is a read operation that reads x after op_2.

(b) *read-write* dependency: $op_1 \rightarrow op_2$, if op_1 is a read operation that reads the data item x, and op_2 is a write operation that overwrites the value of x that op_1 has read.

(c) *write-write* dependency: $op_1 \rightarrow op_2$, if both op_1 and op_2 are write operations on the same data item x, and op_2 overwrites the value written by op_1.

The dependency order defines the relative ordering of read and write operations that are causally related. It requires that reads respect the order of causally related writes. Under causality, all operations that could have influenced one operation must be visible before the operation takes effect. Implementation of causal consistency usually involves dependency

tracking [55, 60, 61]. Dependency tracking associated with each operation is employed to record meta-information for reasoning about the causality. Each process server reads from their local data items and determines when to apply the newer writes to update the local stores based on the dependency tracking.

4.2.3 Ordering Consistency

Instead of merely ensuring partial orderings between causality dependent operations, ordering consistency is an enhanced variation of causal consistency ensuring global ordering of operations. Ordering consistency provides the monotonicity guarantee of both read and write operations to each data item.

(a) The "monotonic writes" guarantee ensures write operations being applied in the identical order on all nodes.

(b) The "monotonic reads" guarantee ensures that reads only see progressively newer versions of data on each node.

The "monotonic writes" guarantee can be enforced by ensuring that write operation can be accepted only if all writes made by the same user are incorporated in the same node [62]. It can be achieved by designating one node as the primary node for every record; and then all updates to that record are first directing to the primary node. The primary node orders operations by assigning them monotonically increasing sequence numbers. All update operations, together with their associated sequence numbers, are then propagated to non-primary nodes by subscribing them to a queue ensuring updates are delivered successfully. In the case that the primary node fails, one of the non-primary nodes is elected to act as the new primary node [8, 55, 63].

From the analysis above, causal consistency is stronger than eventual consistency. Further, it is not difficult to see that ordering consistency is stronger than causal consistency in that a system that guarantees ordering consistency also guarantees causal consistency, but not vice versa, since the causal consistency does not prevent conflicting updates [61]. If we use the $>$ symbol to represent the stronger relationship among two consistency models, the following demonstrates that all the three consistency models form a linear order:

$$ordering\ consistency > causal\ consistency >$$
$$eventual\ consistency$$

More generally, if we consider each operation in BASE as a single operation transac-

---

[20]By the view of single operation transaction, we mean "start transaction" and "commit" are added before and after each and every database operation.

tion[20], the operation schedule in BASE can have an equivalent schedule in the view of single operation transaction in ACID. Considering the bounded staleness [64, 65] for values observed by read, the ordering consistency permits possibly stale of data for read at low cost; however, the read committed guarantees that every read observes the most recent and consistent value of data committed before the start of the transaction. We may conclude the stronger relationship among all consistency models in both ACID and BASE as:

$$serializability > snapshot\ isolation$$
$$> read\ committed > ordering\ consistency >$$
$$causal\ consistency > eventual\ consistency$$

4.3   Consistency Model Taxonomy

Based on the discussion of consistency models, we categorize the implementation of different systems into the taxonomy as shown in Figure 8. The classification is based on our ensuing analysis.

Spanner [14, 44], Megastore [13] and Spinnaker [66] provide one-copy serializability with a two-phase commit and Paxos-based protocol [67]. Megastore and Spinnaker provides serializable pessimistic transactions using strict two-phase locking protocol. Spanner adopts strong timestamp semantics. Every transaction is assigned a commit timestamp, and these timestamps allow Spanner to correctly determine whether a state is sufficiently up-to-date to satisfy a read. Paxos protocol ensures that data will be available as long as the majority of the replicas are alive. To support transactions across multiple sites, the two-phase commit protocol and Paxos are usually combined, such as MDCC [68], 2PC-PCI [61], Paxos-CP [69], and a group of engines have been implemented, such as Chubby [70] and ZooKeeper [71]. Rubato DB [4] supports serialization by introducing a formula protocol that is a variation of the multiversion timestamp concurrency control protocol. The formula protocol reduces the overhead of conventional implementation by using formulas rather than multiple versions. A dynamic timestamp ordering mechanism is also used to increase concurrency and reduce unnecessary blocking.

VoltDB and H-Store [5, 72, 73] support the SQL transaction execution through the stored procedure. By initiating a global order before execution, all nodes can asynchronously execute the stored procedures serially to completion with the same order. Furthermore, H-Store and VoltDB perform the sequential execution in a single-threaded manner without any support for concurrency. The combination of above mechanisms makes the transaction execution in those systems resemble a single oper-
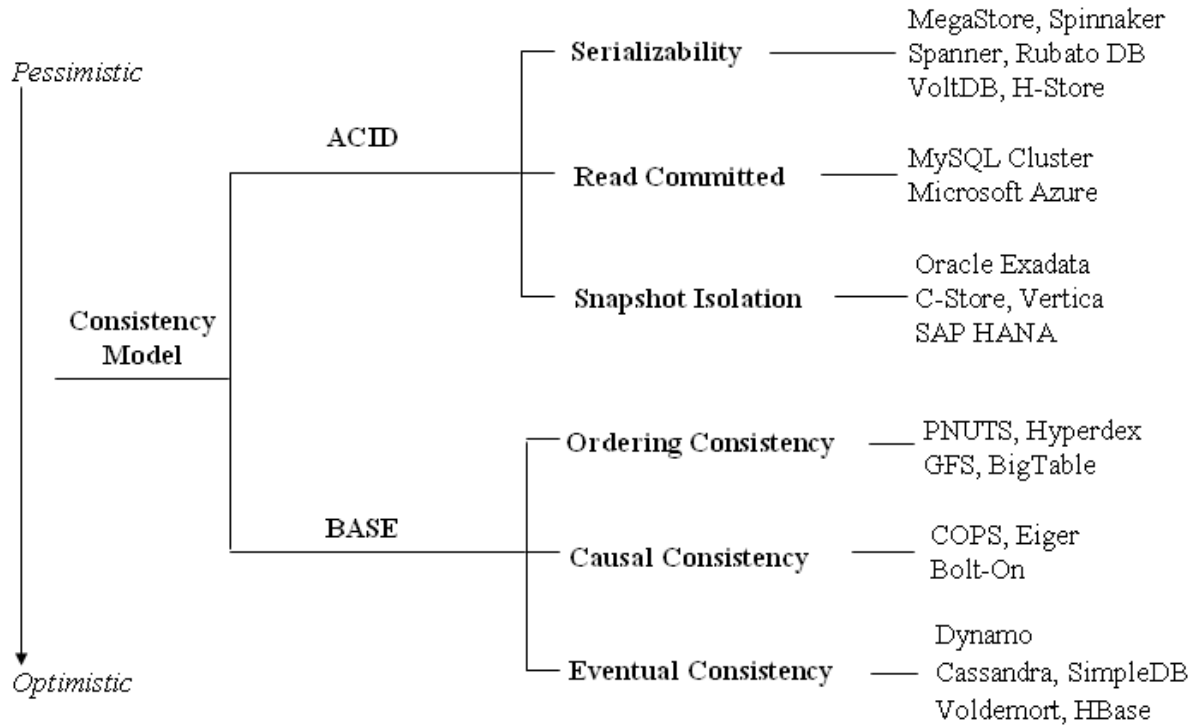
Fig. 8. Taxonomy of Consistency Model

ation call.

MySQL Cluster and Microsoft Azure [3] combined traditional read committed with the master-slave mode synchronization. Exadata isolates read-only transactions using snapshot isolation. High water mark with low-overhead mechanism is introduced for keeping track of the value in multi-replica environment in C-Store [6]. SAP HANA [50] relies on MVCC as the underlying concurrency control mechanism to synchronize multiple writers, and provide distributed snapshot isolation.

Hyperdex [74] provides ordering consistency with a chaining structure, in which nodes are arranged into a value-dependent chain. The head of the chain handles all write operations and dictates the total order on all updates to the object. Each update flows through the chain from the head to the tail, and remains pending until an acknowledgement of commit received from the tail. Read operations can be dealt with by different nodes as long as the latest committed value of data can be obtained on that node.

Yahoo! PNUTS [8] provides a per-record timeline consistent model that preserves ordering consistency. PNUTS introduces a pub/sub message broker, which takes charge of receiving updates from a master node and sending them to other nodes in an identical sequence order.

GFS [11] and Bigtable [12] both use Chubby [70], a distributed locking mechanism for distributed node coordination. They rely on Chubby with a lease agreement to apply mutations to each chunk in the same order. The global mutation order is defined by the lease grant order.

COPS [55, 63] and Eiger [61] track dependencies on versions of keys or operations to enforce causal consistency. An operation does not take effect until verifying that the operation's dependencies are satisfied.

Bolt-on [60] provides a shim layer that upgrades the eventual consistency of an underlying general-purpose data store to the causal consistency for the application. Bolt-on sets a generalized, declarative specification of causal cut as the criteria to determine which writes are made visible to clients.

Dynamo [7], Voldemort and Cassandra [9] provide eventual consistency for allowing applications with "always writeable" property, that is, write operations can always be accepted by any node. Vector clock, also named as version vector, is associated with data to determine the eventual consistent state during reconciliation.

5 Implementation Principles for Large-Scale System

So far, we have built the taxonomies for the state-of-the-art large-scale data management systems based on the data model, the system architecture and the consistency model. In this section, we will discuss and analyze the scalability limitation for different designs and implementations, and provide our principles for the implementation of the next generation large-scale data management systems.

5.1 Data Model Implementation

To achieve high scalability, systems need to distribute data to different nodes. The simplest way for the data distribution is to deploy individual tables at different nodes. However, since concurrent transactions usually access different portions of one table, thus the table partition can improve the performance by parallel execution on a number of nodes. In addition, when processing a single query over a large table, the response time can be reduced by distributing the execution across multiple nodes [20]. Partitioning can be either horizontal or vertical, and systems require a close integration between data model and data partition.

Physical level. For the row-oriented layout, horizontal partitioning is properly used

where each partition contains a subset of the rows and each row is in exactly one partition. For column-oriented or hybrid layout, vertical or mixed partitioning can be applied. The fine-grained mixed partitioning is often implemented by vertically partitioning columns of a table into frequent and static column groups. Columns frequently accessed together are clustered into the same frequent column group, while columns seldom accessed are gathered into static column groups [24, 72]. In addition, columns with large-size data are separated independently to take the advantage of the compression benefits of column stores [26].

Conceptual level. Regarding to the conceptual schema, numerous NoSQL high scalable systems such as key-value stores [7] and Bigtable-like stores [12] represent a recent evolution of making trade-off between scalability and schema complexity. These NoSQL systems adopt variant of schema-less unstructured data for large-scale data applications. However, some NewSQL systems relatively to NoSQL seek to provide the high scalability and throughput same as NoSQL while still preserving the high level of structured data model [5, 13, 14, 20].

To achieve high scalability of a structured data store, the partitioning must be based on the relational schema and query workloads in order to minimize the contention. Inappropriate partitioning may cause data skew. The skewed data will decline the response time and generate hot nodes, which easily become a bottleneck throttling the overall performance and scalability [75, 76]. For example, to obtain the optimal partitioning, the schema of OLTP workload always transits to a tree structure. Tuples in every descendent table are partitioned according to the ancestor that they descend from. As a consequence, data accessed within a transaction will be located in the same data node [24, 72].

Based on the analysis above and the taxonomy in Section 2, we believe that:

1. The data model with lower conceptual level can simplify the design and implementation for scale-out capability, but data model with higher conceptual level is not an obstacle of scalability as long as an optimal partitioning strategy is applied according to the schema.

2. The hybrid storage is the most flexible to leverage the advantages of row stores and column stores. And hybrid storage facilitates the close integration of data modeling and data partitioning.

5.2 Architecture Scalability

Systems based on shared-resources infrastructure can only be scaled up with inevitable bound due to resources contention and limitation. For example, the internal bus bandwidth, the number of CPUs, the hardware coordination mechanism [30], the context switches [73, 77], all can inhibit scalability.

In the sharding on shared-nothing architecture, local resources serve local processors; thus it overcomes the disadvantage of shared-resource infrastructure. The shared-nothing design is intended to support scaling out smoothly by involving new computation nodes. The centralized topology uses one dedicated centralized coordinator for managing system-wide membership state. Thus, it may suffer the single point failure and the central node is susceptive to become the bottleneck. The scalability of the decentralized peer-to-peer topology can excel that of the centralized one [7, 9].

Though systems on sharding on shared-nothing architecture are desired to achieve ultimate scalability, and have been proven to scale really well into tens of hundreds of nodes. However, there are very few known data management system deployments with thousands of nodes [21, 49, 44, 47]. One reason is that at such scale, node failure becomes rather common, while most sharding systems such as parallel relational databases are not fault tolerant enough [21, 44].

SEDA/MapReduce architecture is able to scale to thousands of nodes due to its superior scalability, fault tolerance and flexibility. In particular,

(a) The architecture facilitates the use of shared-nothing infrastructure as a scalable platform for applications, which is easy to scale out the critical components by adding more computing resources. Computing resources can be distributed flexibly to wherever more computation power is needed to eliminate bottleneck.

(b) The decomposition of a complex database management system into various stages connected by queues not only enables modularity and reuse of stage modules, but also allows two types of parallel data processing: pipeline parallelism and partition parallelism.

(c) Since a data flow is distributed not only between multiple server instances (usually running on different physical nodes) but also between different staged modules, multiple identical stages can be executed on multiple cluster machines with data replication in order to achieve high avail-

ability for fault tolerance.

(d) The architecture easily supports data partition. Different partitions can be distributed to different staged modules on multiple computing nodes. Different stage modules are communicated by passing operation. The downstream module can start consuming operation when the producer module passes the operation.

A large number of system applications based on this architecture are such good examples:

1. Google uses the MapReduce framework internally to process more than 20PB datasets per-day, achieving the ability to sort 1 PB data using 4,000 commodity servers [15].

2. Hadoop at Yahoo! is assembled on 3,000 nodes with 16PB raw disk [16].

3. At Facebook, Hive [19] forms the storage and analytics system infrastructure that stores 15PB data and process 60TB new data everyday with thousands of nodes [39].

4. Microsoft Dryad conducts data mining style operations to tens of perabytes data using a cluster of around 1800 computers [49].

5. Google Dremel manages trillion-record, multi-perabyte datasets, running on 1000-4000 nodes with near-linear scalability[47].

Based on the above empirical evidence and the system taxonomy in Section 3, we can have that the scalability capacity of these different architectures can be ordered as:

$$SEDA/MapReduce >> Sharding\ Decentralized$$
$$> Sharding\ Centralized >> Shared\text{-}Disk\ MPP$$
$$> Shared\text{-}Memory\ SMP.$$

Thus, the SEDA/MapReduce achitecture is the most suitable for large-scale data management systems.

5.3   Consistency Model Implementation

We propose a consistency model taxonomy in which the model in a higher level provides stricter guarantees than the model in a lower level. Now we discuss the relationship between consistency and scalability, and especially how the implementation of different consistency models will affect the scalability.

5.3.1   Protocols for ACID

The common implementation to provide serializability is based on distributed two-phase locking (2PL) protocol [13, 66]. However, the locking-based protocol adds overhead to each

data access due to the manipulation to acquire and release locks, and it limits concurrency and scalability in case of conflicting accesses, and adds overheads due to deadlock detection and resolution [78, 79, 80]. Another similar pessimistic concurrency control protocol implementation is based on distributed serialization graph testing (SGT), which characterizes conflict serializability via the absence of cycles in the conflict graph [79]. The limitation of this implementation is closely related to the problem of testing a distributed graph for cycles, which also arises the deadlock detection issue. Thus, transactions executed by a distributed large-scale database should not cause distributed deadlocks that are rather difficult and expensive to deal with.

Optimistic protocol is lock-free assuming that conflicts between transactions are rare. However, to guarantee that the validation phase can produce consistent results, a global order checking is required which will degrade performance heavily in large-scale distributed data systems [14]. Additionally, in the presence of slower network connection, more transactions may crowed into the system causing excessively high chances of rollbacks [78]. Therefore, the timestamp-based concurrency control is the most suitable for large-scale data management systems, and Google Spanner [14]

based on "True Time" is such a good example.

The communication latency caused by various protocol implementations (e.g. two-phase commit, three-phase commit, Paxos, etc.) can limit the scalability due to the overhead caused by multiple network round-trips [81]. Though a protocol implementation performs well within a small scale cluster, however, it may severely limit the scalability of large-scale systems, since the availability and coordination overheads become worse as the number of nodes increases [82, 83].

Constraining the scope of transactions is one typical way to minimize the high-latency communication overhead. The restriction of transactions alleviates the transaction coordination protocol and reduces message delays. A list of systems such as Azure [3], MySQL Cluster, Megastore [13], Sinfonia [84] and H-Store [5, 73] only support restricted transactions that can be executed in parallel to completion without requiring communication with other repositories or any commit vote phase. This restrictive scope is reasonable for the applications where data can be well deployed, so that distributed transactions will be very rare in such cases.

5.3.2 Alternative for BASE

BASE can achieve high scalability much easier than ACID, but it has its own potential disadvantages.

- Firstly, *eventual consistency* makes only liveness rather than safety guarantee, as it merely ensures the system to converge to a consistent state in the future [55].

- Secondly, the *soft state* presents challenges for developers, which requires extremely complex and error-prone mechanisms to reason the correctness of the system state at each single point in time [7, 8, 44].

- Thirdly, additional restriction is required for the *soft state* to converge to the *eventual consistency* [56, 57, 58].

We first use a simple example to demonstrate deficiencies of *BASE* listed above. Consider one data item[21] with three columns (fields) L, S and H; and the following three atomic operations on the data item that are non-commutative[22]:

1. $a(x)$: $W_a(L = L \times (1+x\%), S = S \times (1+x\%), H = H \times (1+x\%))$;

2. $b(y)$: $W_b(L = L + y, S = S + y, H = H + y)$;

3. *check*: $R_c(L, S, H)$, $Assert(L = S = H)$.

Naturally, we assume that the state of the system transits from one consistent state to another if and only if it is caused by the completion of an atomic operation. We also assume that, due to column partitioning and grid partitioning for scalability, columns L, S, H are partitioned into different tables and distributed to three different nodes N_1, N_2, N_3, respectively. For simplicity, we assume initial values in the columns L, S, H are 100.

Consider a schedule
$S = \{a(20), b(10), check, check\}$,
as shown in Table 1, with overlaps as follows:

1. $a(20)$ is incomplete when $b(10)$ starts at t_2.

2. The first *check* is issued at t_3 before $a(20)$ is done.

In such schedule, both *checks* return inconsistent states. The first *check* suffered from *soft state* that requires users to reason about the correctness. The second *check*, even though being issued after t_4, can not guarantee the even-

---

[21]For simplicity, we consider only one data item but all the discussions are valid for a set of data items.

[22]Commutative operations such as increment/decrement can exchange the execution order without affecting the result.

schedule				value		
time	Node N_1	Node N_2	Node N_3	L	S	H
t_1	$W_{a1}(L \times (1 + 20\%))$	$W_{a2}(S \times (1 + 20\%))$		120	120	100
t_2	$W_{b1}(L + 10)$	$W_{b2}(S + 10)$	$W_{b3}(H + 10)$	130	130	110
t_3	$R_{c1}(L = 130)$	$R_{c2}(S = 130)$	$R_{c3}(H = 110)$			
t_4			$W_{a3}(H \times (1 + 20\%))$	130	130	132
t_5	$R_{c1}(L = 130)$	$R_{c2}(S = 130)$	$R_{c3}(H = 132)$			

Table 1.   Schedule $S = \{a(20), b(10), check, check\}$ with W/W and R/W overlaps. W_{ai}, W_{bi}, R_{ci} represent the corresponding actions of operations a, b and *check* on node N_i, $i = 1, 2, 3$ respectively

tual consistent state. Thus additional restriction is required to achieve reconciliation.

In order to resolve the weakness of BASE, **BASIC** properties [85] are proposed, standing for **B**asic **A**vailability, **S**calability and **I**nstant **C**onsistency.

- Basic Availability: the system can response to all continuously operations.

- Scalability: the system is able to scale out by adding more resources for increasing workloads.

- Instant Consistency: all data seen by reads reflects a consistent state of the system. i.e., each read returns the result that reflects all write operations that

have received response successfully prior to the read.

The *BASIC* properties aim to achieve the availability and scalability same as *BASE* properties while getting rid of the *soft state* by introducing *instant consistency*.

Intuitively, a schedule satisfies *instant consistency* if its equivalent schedule in the view of single operation transaction is serializable. Because of the write latency and data distribution, it is very difficult, if not impossible, to use only serial schedules in large-scale data management systems. We can define instant consistency based on snapshot as following:

Definition 1. *A* snapshot *of a database represents a complete copy of all the data items*

updated by a serial schedule. A schedule of a set of operations satisfies instant consistency *if any READ operation reads from a snapshot.*

Rubato DB [4] implemented a timestamp-based formula protocol that guarantees *BASIC* properties. It demonstrates that *BASIC* can be achieved with linear scalability, and the performance decline induced by *BASIC* is acceptable comparing with the extra efforts needed to manipulate inconsistent soft states of BASE. That is, *BASIC* pays a reasonable price for a higher consistency than *BASE.*

Based on the discussion above and the classification in Section 4, we believe that:

1. The weak consistency model like BASE can achieve high scalability much easier than the strong consistency model like ACID. But the strong consistency model does not hinder the scale-out capability as long as a proper implementation (or reasonable restriction) is developed.

2. The timestamp-based concurrency control protocols are the best suitable to scale out.

3. Consistency models stronger than BASE, such as BASIC, are desirable for application developers.

6 Conclusion

In this survey, we have investigated, categorized and studied several critical aspects of large-scale data management systems. These systems have several unique characteristics mainly including scalability, elasticity, manageability, and low cost-efficiency. We have first enumerated various data models on physical layouts and conceptual representations. And then, we focused on the design and implementation of system architectures. We have developed architecture taxonomies for prevailing large-scale database systems to classify the common architecture designs, and provide comparison of the capability for scale-out. We then compared two categories of the consistency models and classified prevailing systems according to the respective taxonomies. With this mapping, we have gained insights into the trade-off between consistency and scalability.

Based on our taxonomies and characterization, we identify the principles for implementation of large scale data management systems including:

• The hybrid storage layout can facilitate the close integration of data modeling and data partitioning.

• The SEDA/MapReduce architecture is the optimal to achieve high scalability.

- The timestamp-based protocol is the most suitable to scale out.

- The strong consistency model is not an obstacle for scalability with a proper implementation (or reasonable restriction).

- BASE is so weak that a stronger consistency model like BASIC is desirable.

To conclude, this work delves deeper to lay down a comprehensive taxonomy framework that, not only serves as a direction of analyzing the large-scale data management systems for the big data application, but also presents references and principles for which future implementation and efforts need to be undertaken by developers and researchers.

References

[1] Agrawal D, Das S, Abbadi A. Big data and cloud computing: current state and future opportunities. In *Proc. the 14th International Conference on Extending Database Technology*, March 2011, pp 530–533.

[2] Glogor G, Silviu T. Oracle exalytics: Engineered for speed-of-thought analytics. *Database Systems Journal*, 2011, 2(4): 3-8.

[3] Campbell D, Kakivaya G, Ellis N. Extreme scale with full SQL language support in Microsoft SQL Azure. In *Proc. the 2010 ACM SIGMOD International Conference on Management of Data*, June 2010, pp 1021–1024.

[4] Yuan L, Wu L, You J, Chi Y. Rubato DB: A highly scalable staged grid database system for OLTP and big data applications. In *ACM International Conference on Information and Knowledge Management*, November 2014.

[5] Kallman R, Kimura H, Natkins J et al. H-store: a high-performance, distributed main memory transaction processing system. In *Proc. the 34th International Conference on Very Large Data Bases*, August 2008, pp 1496–1499.

[6] Stonebraker M, Abadi D, Batkin A et al. C-store: a column-oriented DBMS. In *Proc. the 31st International Conference on Very Large Data Bases*, August 2005, pp 553–564.

[7] DeCandia G, Hastorun D, Jampani M et al. Dynamo: Amazon's highly available key-value store. In *Proc. the 21st ACM SIGOPS Symposium on Operating Systems Principles*, October 2007, pp 205–220.

[8] Cooper BF, Ramakrishnan R, Srivastava U et al. PNUTS: Yahoo!'s hosted data serving platform. In *Proc. the 34th International Conference on Very Large Data Bases*, August 2008, pp 1277–1288.

[9] Lakshman A, Malik P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 2010, 44(2): 35-40.

[10] Joshi A, Sam H, Charles L. Oracle NoSQL database-scalable, transactional key-value store. In *the 2nd International Conference on Advances in Information Mining and Management*, October 2012, pp 75–78.

[11] Ghemawat S, Gobioff H, Leung S. The Google file system. In *Proc. the 19th ACM Symposium on Operating Systems Principles*, December 2003, pp 29–43.

[12] Chang F, Dean J, Ghemawat S et al. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 2008, 26(2): 1–26.

[13] Baker J, Bond C, Corbett J et al. Megastore: providing scalable, highly available storage for interactive services. In *the 5th Conference on Innovative Data Systems Research*, January 2011, pp 223–234.

[14] Corbett J, Dean J, Epstein M et al. Spanner: Google's globally-distributed database. In *Proc. the 10th USENIX Conference on Operating Systems Design and Implementation*, October 2012, pp 251–264.

[15] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Communication ACM* 2008, 51(1): 107–113.

[16] Shvachko K, Kuang H, Radia S, Chansler R. The Hadoop distributed file system. In *Proc. the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, May 2010, pp 1–10

[17] Olston C, Reed B, Srivastava U et al. Pig latin: a not-so-foreign language for data processing. In *Proc. the 2008 ACM SIGMOD International Conference on Management of Data*, June 2008, pp 1099–1110.

[18] Chaiken R, Jenkins B, Larson P et al. SCOPE: easy and efficient parallel processing of massive data sets. In *Proc. the 34th International Conference on Very Large Data Bases*, August 2008, pp 1265–1276.

[19] Thusoo A, Sarma J, Jain N et al. Hive: a warehousing solution over a map-reduce framework. In *Proc. the 35th International Conference on Very Large Data Bases*, August 2009, pp 1626–1629.

[20] Cohen J, Dolan B, Dunlap M, Hellerstein JM, Welton C. MAD skills: new analysis practices for big data. In *Proc. the 35th International Conference on Very Large Data Bases*, August 2009, pp 1481–1492.

[21] Abouzeid A, Bajda-Pawlikowski K, Abadi D et al. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *Proc. the 35th International Conference on Very Large Data Bases*, August 2009, pp 922–933.

[22] Abadi D, Madden S, Hachem N. Column-Stores vs. Row-Stores: How different are they really? In *Proc. the 2008 ACM SIGMOD International Conference on Management of Data*, June 2005, pp 967–980.

[23] Ramamurthy R, DeWitt D, Su Q. A case for fractured mirrors. In *Proc. the 29th International Conference on Very Large Data Bases*, August 2003, pp 89–101.

[24] Grund M, Krüger J, Plattner H, Zeier A, Cudre-Mauroux P, Madden S. HYRISE: a main memory hybrid storage engine. In *Proc. the 36th International Conference on Very Large Data Bases*, November 2010, pp 105–116.

[25] Hankins R, Patel J. Data morphing: an adaptive, cache-conscious storage technique. In *Proc. the 29th International Conference on Very Large Data Bases*, September 2003, pp 417–428.

[26] Abadi D, Madden S, Ferreira M. Integrating compression and execution in column-oriented database systems. In *Proc. the 2006 ACM SIGMOD International Conference on Management of Data*, June 2006, pp 671–682.

[27] Boncz P, Grust T, Keulen M, Manegold S, Rittinger J, Teubner J. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proc. the 2006 ACM SIGMOD International Conference on Management of Data*, June 2006, pp 479–490.

[28] Manegold S, Kersten ML, Boncz P. Database architecture evolution: mammals flourished long before dinosaurs became extinct. In *Proc. the 35th International Conference on Very Large Data Bases*, August 2009, pp 1648–1653.

[29] Ailamaki A, DeWitt D, Hill M, Skounakis M. Weaving relations for cache performance. In *Proc. the 27th International Conference on Very Large Data Bases*, September 2001, pp 169–180.

[30] Poess M, Nambiar R. Large scale data warehouses on grid: Oracle database 10g and HP proliant servers. In *Proc. the 31st International Conference on Very Large Data Bases*, September 2005, pp 1055–1066.

[31] Gibson G, Van Meter R. Network attached storage architecture. *Communications of the ACM*, 2000, 43(11): 37-45.

[32] Bridge W, Joshi A, Keihl M et al. The Oracle universal server buffer. In *Proc. the 23rd International Conference on Very Large Data Bases*, August 1997, pp 590–594.

[33] Lahiri T, Srihari V, Chan W et al. Cache fusion: extending shared-disk clusters with shared caches. In *Proc. the 27th International Conference on Very Large Data Bases*, September 2001, pp 683–686.

[34] Birman K. The promise, and limitations, of gossip protocols. In *SIGOPS Operation System Review*, October 2007, pp 8–13.

[35] Welsh M, Culler D, Brewer E. SEDA: an architecture for well-conditioned, scalable internet services. In *Proc. the 18th ACM Symposium on Operating Systems Principles*, October 2001, pp 230–243.

[36] Harizopoulos S, Ailamaki A. A case for staged database systems. In *Proc. the 1st Conference on Innovative Data Systems Research*, January 2003, pp 101–112.

[37] Condie T, Conway N, Alvaro P et al. Online aggregation and continuous query support in Mapreduce. In *Proc. the 2010 ACM SIGMOD International Conference on Management of Data*, June 2010, pp 1115–1118.

[38] Verma A, Cho B, Zea N et al. Breaking the Mapreduce stage barrier. *Journal of Cluster Computing*, 2013, 16(1): 191-206.

[39] Thusoo A, Shao Z, Anthony S et al. Data warehousing and analytics infrastructure at Facebook. In *Proc. the 2010 ACM SIGMOD International Conference on Management of data*, June 2010, pp 1013–1020.

[40] Thomson A, Diamond T, Weng S et al. Calvin: fast distributed transactions for partitioned database systems. In *Proc. the 2012 ACM SIGMOD International Con-*

ference on Management of Data, May 2012, pp 1–12.

[41] Gummadi K, Gummadi R, Gribble S et al. The impact of DHT routing geometry on resilience and proximity. In *Proc. the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, August 2003, pp 381–394.

[42] Karger D, Lehman E, Leighton T et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. the 29th Annual ACM Symposium on Theory of Computing*, May 1997, pp 654–663.

[43] Alvisi L, Malkhi D, Pierce E et al. Fault detection for Byzantine quorum systems. *IEEE Transaction Parallel Distribute System*, 2001, 12(9): 996–1007.

[44] Shute J, Vingralek R, Samwel B et al. F1: A distributed SQL database that scales. In *Proc. the 39th International Conference on Very Large Data Bases*, August 2013, pp 1068–1079.

[45] Zaharia M, Mosharaf C, Michael J et al. Spark: cluster computing with working sets. In *Proc. the 2nd USENIX Conference on Hot Topics in Cloud Computing*, June 2010, pp 10–12.

[46] Xin R, Josh R, Zaharia M et al. Shark: SQL and rich analytics at scale. In *Proc. the 2013 International Conference on Management of Data*, June 2013, pp 13–24.

[47] Melnik S, Gubarev A, Long J et al. Dremel: interactive analysis of web-scale datasets. In *Proc. the 36th International Conference on Very Large Data Bases*, September 2010, pp 330–339.

[48] Lewis P, Bernstein A, Kifer M. Databases and transaction processing: an application-oriented approach. Addison-Wesley Longman Publishing. 2002, chapter 18, pp 764–773

[49] Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 2007, 41(3): 59-72.

[50] Vishal S, Franz F, Lehner W et al. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proc. the 2012 ACM SIGMOD International Conference on Management of Data*, May 2012, pp 731–742.

[51] Berenson H, Bernstein P, Gray J et al. A critique of ANSI SQL isolation levels. In *Proc. the 1995 ACM SIGMOD International Conference on Management of Data*, May 1995, pp 1–10.

[52] Bornea M, Hodson O, Elnikety S, Fekete A. One-copy serializability with snapshot isolation under the hood. In *Proc. the 2011 IEEE 27th International Conference on Data Engineering*, April 2011, pp 625–636.

[53] Lin Y, Kemme B et al. Middleware based data replication providing snapshot isolation. In *Proc. the 2005 ACM SIGMOD International Conference on Management of Data*, July 2005, pp 419–430.

[54] Pritchett D. BASE: An ACID alternative. *ACM Queue*, 2008, 6(3): 48–55.

[55] Lloyd W, Freedman M, Kaminsky M et al. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proc. the 23rd ACM Symposium on Operating Systems Principles*, October 2011, pp 401–416.

[56] Roh H, Jeon M, Kim J. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel Distributed Computing*, 2011, 71(3): 354–368.

[57] Shapiro M, Preguiça N, Baquero C, Zawirski M. Conflict-free replicated data types. In *Proc. the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, June 2011, pp 386–400.

[58] Vogels W. Eventually consistent. *ACM Queue*, 2008, 6(6): 14–19.

[59] Ahamad M, Neiger G, Burns J et al. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 1995, 9(1): 37–49.

[60] Bailis P, Ghodsi A, Heller J, Stoica I. Bolt-on causal consistency. In *Proc. the 2013 ACM SIGMOD International Conference on Management of Data*, June 2013, pp 761–772.

[61] Lloyd W, Freedman M, Kaminsky M. Stronger semantics for low-latency geo-replicated storage. In *Proc. the 10th USENIX Conference on Networked Systems Design and Implementation*, April 2013, pp 313–328.

[62] Saito Y, Shapiro M. Optimistic replication. *ACM Computer Survey*, 2005, 1(37): 42–81.

[63] Burckhardt S, Leijen D et al. Eventually consistent transactions. In *Proc. the*

21st European Conference on Programming Languages and Systems, June 2010, pp 67–86.

[64] Bailis P, Venkataraman S., Franklin M et al. Probabilistically bounded staleness for practical partial quorums. In *Proc. the 38th European Conference on Programming Languages and Systems*, August 2012, pp 776–787.

[65] Cipar J, Ganger G, Keeton K et al. Lazy-Base: trading freshness for performance in a scalable database. In *Proc. the 7th ACM European Conference on Computer Systems*, April 2012, pp 169-182.

[66] Rao J, Shekita EJ, Tata S. Using Paxos to build a scalable, consistent, and highly available datastore. In *Proc. the 37th International Conference on Very Large Data Bases*, January 2011, pp 243–254.

[67] Chandra TD, Griesemer R, Redstone J. Paxos made live: An engineering perspective. In *Proc. the 26th Annual ACM Symposium on Principles of Distributed Computing*, August 2007, pp 398–407.

[68] Kraska T, Pang G, Franklin M et al. MDCC: Multi-data center consistency. In *Proc. the 8th ACM European Conference on Computer Systems*, April 2013, pp 113–126.

[69] Patterson S, Elmore AJ, Nawab F et al. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. In *Proc. the 38th International Conference on Very Large Data Bases*, July 2012, pp 1459–1470.

[70] Burrows M. The chubby lock service for loosely-coupled distributed systems. In *Proc. the 7th Symposium on Operating Systems Design and Implementation*, November 2006, pp 335–350.

[71] Hunt P, Konar M, Junqueira F. ZooKeeper: wait-free coordination for internet-scale systems. In *Proc. the 2010 USENIX Conference on USENIX Annual Technical Conference*, June 2010, pp 11-11.

[72] Jones E, Abadi D, Madden S. Low overhead concurrency control for partitioned main memory databases. In *Proc. the 2010 ACM SIGMOD International Conference on Management of Data*, June 2010, pp 603–614.

[73] Stonebraker M, Madden S, Abadi D. The end of an architectural era: (it's time for a complete rewrite). In *Proc. the 33rd Inter-*

national Conference on Very Large Data Bases, September 2007, pp 1150–1160.

[74] Escriva R, Wong B, Sirer E. Hyper-Dex: A distributed, searchable key-value store. *SIGCOMM Computer Communication Review*, 2012, 42(4): 25–36.

[75] Kossmann D, Kraskan T, Loesing S. An evaluation of alternative architectures for transaction processing in the cloud. In *Proc. the 2010 ACM SIGMOD International Conference on Management of Data*, June 2010, pp 579–590.

[76] Xu Y, Kostamaa P, Zhou X, Chen L. Handling data skew in parallel joins in shared-nothing systems. In *Proc. the 2008 ACM SIGMOD International Conference on Management of Data*, June 2008, pp 1043–1052.

[77] Johnson R, Pandis I, Hardavellas N et al. Shore-MT: a scalable storage manager for the multi-core era. In *Proc. the 12th International Conference on Extending Database Technology: Advances in Database Technology*, March 2009, pp 24–35.

[78] Larson P, Blanas S et al. High-performance concurrency control mechanisms for main-memory databases. In *J. Comput. Sci. & Technol., Aug.. 2014, , Proc. the 37th International Conference on Very Large Data Bases*, August 2011, pp 298–309.

[79] Özsu M. Principles of Distributed Database Systems (3rd Edition). 2011, Chapter 11, pp 387–394. Springer.

[80] Weikum G, Vossen G. Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. 2001, pp 676–686. Morgan Kaufmann Publishers.

[81] Helland P. Life beyond distributed transactions: an apostate's opinion. In *3rd Biennial Conference on Innovative Data Systems Research*, January 2007, pp 132–141.

[82] Yu H, Vahdat A. Minimal replication cost for availability. In *Proc. the 21st Annual Symposium on Principles of Distributed Computing*, July 2002, pp 98–107.

[83] Yu H, Vahdat A. The costs and limits of availability for replicated services. *ACM Transaction Computer System*, 2006, 1(24): 70-113.

[84] Aguilera M, Merchant A, Shah M et al. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Transaction Computer System*, 2009, 5(27): 1–48.

[85] Wu L, Yuan L, You J. BASIC, an alternative to BASE for large-scale data management system. In *2014 IEEE International Conference on Big Data*, October 2014.