To appear in Journal of Computer Science and Technology

Survey and Taxonomy of Large-scale Data Management Systems for Big Data Applications

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

Department of Computing Science, University of Alberta, Edmonton, AB, Canada

Abstract. Today, data is flowing into various organizations at an unprecedented scale in the world. 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 enterprises.

In this article, we analyze the large-scale data management systems in depth and develop comprehensive taxonomies for various critical aspects covering storage model, system architecture and consistency model. We map the prevailing highly scalable systems to the proposed taxonomy, not only to classify the common techniques but also to provide a basis for analyzing current system scalability limitations. These limitations indicate some possible directions that future efforts need to be undertaken by researchers.

Keywords: Data storage; System Architecture; Consistency; 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 (Agrawal et al, 2011).

The pursuit for tackling the challenges posed by management of big data

Received xxx
Revised xxx
Accepted xxx

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 phenomena is NoSQL (Not Only SQL) movement that began in early 2009 and is evolving rapidly. Numerous NoSQL systems, based on MapReduce and Hadoop (Dean and Ghemawat, 2008; Shvachko et al, 2010), utilize a large collection of commodity servers to provide high scalability for big data management application.

In this work, we provide a comprehensive study of the state-of-the-art large-scale data management systems for 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 system limitations and ensuring a successful system transition 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 volume and a large number of concurrent users require substantially throughput raising for the systems.
- *Velocity*: Data is flowing in at unprecedented speed and needs to be dealt with in a timely manner.
- *Variety*: Data comes in all types of formats, from structured relational data to unstructured data.
- *Veracity*: Inconsistency or uncertainty of data, due to the quality of 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 system. 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 system. For example, Google responds to the Web-scale storage challenges by developing a family of systems. Google File System (GFS) (Ghemawat et al, 2003) 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 (Chang et al, 2010) is a hybrid data storage model built on GFS, commonly used as the interface for MapReduce framework (Dean and Ghemawat, 2008). Megastore (Baker et al, 2011) and Spanner (Corbett et al, 2012) are two systems over BigTable layer. Megastore blends the scalability and 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. Taking Google's released papers as guideline, open-source equivalents were developed as well, such as Apache Hadoop MapReduce platform built on the Hadoop Distributed File System (HDFS) (Shvachko et al, 2010). Accordingly, a set of systems with high-level declarative languages, including Yahoo! Pig (Olston et al, 2008), and

Facebook Hive (Thusoo et al, 2009), are realized to compile queries into MapReduce framework before execution on Hadoop platform. In addition to MapReduce and Hadoop, Amazon has found a need for a highly available and scalable distributed key-value data stores providing the reliable and "always-writable" property, leading to the development of Dynamo (DeCandia et al, 2007). An open-source clone of Dynamo, Cassandra (Lakshman et al, 2010), has also been developed by the Apache community.

Systems above are regarded as generalized databases addressing properties of being schema-free, simple-API, horizontally scalable and relaxed consistency. As expected, traditional database vendors such as Oracle, IBM and Microsoft, have developed their own system technologies in response to the similar requirements. Oracle Exadata (Oracle, 2012) and IBM Netezza (Francisco, 2011) delivered high performance by leveraging a massively parallel fashion within a collection of storage cells. Highly parallelized I/O between storage cells and processing units with date filtering were developed to minimize the I/O bottleneck and free up downstream components. Microsoft developed parallel runtime systems including Azure (Campbell et al, 2010) and SCOPE (Chaiken et al, 2008) utilizing specific cluster control with minimal invasion into the SQL Server code base. Some research prototypes, such as H-store (Kallman et al, 2008), later commercialized into VoltDB (VoltDB, 2011), and C-Store (Stonebraker et al, 2005), the predecessor of Vertica (Vertica, 2011), also provided their tentative solutions to the "big data" challenge.

1.2. Synopsis

Having set the stage for the need of large-scale data management system design and implementation, in this survey, we delve deeper to present our insights in the critical aspects of highly scalable systems for big data applications. We give taxonomies from essential aspects as follows:

- *Data storage model.* We capture both key physical and conceptual aspects of data structure for the large-scale data management systems.
- *System Architecture.* The system architecture describes the organization of all database objects and how they work together for data management. We give a comprehensive description of diverse architectures by examining and distinguishing the way how various modules are orchestrated within a system and how data flow control logic is distributed throughout the system.
- *Consistency model.* The consistency model guarantees the possible order and dependency of operations throughout the system, facilitating to simplify the behaviors that must reason about and anomalies that can occur. We investigate progressive consistency levels applied by existing systems, and analyze various consistency models ranging from the weakest to the strongest one.

The remainder of this survey is organized as follows. Section 2 discusses the different design of storage model from two aspects of physical layout and conceptual model. Section 3 undertakes a deep study on the diverse scalable architecture designs and their limitations of scalability. Section 4 presents the different levels of consistency model, the scalability limitations and tradeoff for availability and consistency. Section 5 concludes this survey.

2. Data Model

Data model consists of two essential levels: physical level and conceptual level. The details of how data is stored in the database belong to the physical level of data modeling[1]. 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 for 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 for a particular row is stored sequentially within a single database block. Traditional DBMS towards ad-hoc querying of data tends to choose row-oriented layout.

Column-oriented Layout. Data is organized in a significant deviation of the row-oriented layout. Unlike row-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, tends to adopt the column-oriented layout. It 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, which can reduce I/O significantly (Abadi et al, 2008).

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

Hybrid layout schemes are designed based on different granularities. The most coarse-gained granularity essentially adopts different layout on different replicas like fractured mirrors (Ramamurthy et al, 2009). 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 row-oriented layout while the other one is in column-oriented layout. Fractured mirror can be regarded as a new form of RAID-1, in which the query optimizer decides which replica is the best choice for a corresponding query execution. The fine-grained hybrid schema (Grund et al, 2010; Hankins and Patel, 2003) integrates row and column layouts in the granularity of individual tables. Some parts of the table are stored with row-oriented layout, while other parts apply column-oriented layout. An even finer schema is based on the granularity of disk block. Data in some blocks is aligned by rows while some is aligned by columns (Oracle, 2012). To some extend, we can consider that, *row-oriented layout, column-oriented layout* are special extreme cases of *hybrid-oriented layout*.

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

2.2. Conceptual Level

Obtaining maximum performance requires a close integration between its physical layout and 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. A unstructured data store is at the lowest conceptual level. 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. Operations are limited to those on single key/value pair without cross-references between distinct pairs and thus there is no support for relational schema. 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, providing developers the maximal flexibility to program customized features. Unstructured data stores have gained popularity in the large-scale web services (DeCandia et al, 2007; Ghemawat et al, 2003; Cooper et al, 2008). 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 are 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 application and the database itself, and there can provide some simple query-by-value capability, but the application-based query logic may be complex [cite???]. Because of its schemaless nature, a semi-structured data store can only adapt 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 schemas, 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 set of integrity constraints, and supports SQL as the query language [cite???].

2.3. Data Model Taxonomy

Based on the classification of physical layout and conceptual schema, we analyze currently prevailing database systems and categorize them in an appropriate taxonomy, as demonstrated in Figure 1. This classification is based on our observations outlined below.

Google File System (GFS) (Ghemawat et al, 2003), Hadoop Distributed File System (HDFS) (Shvachko et al, 2010) and Amazon's Dynamo (DeCandia et al, 2007) are the most typical systems belonging to the category with row-oriented physical layout and unstructured conceptual data store. GFS and HDFS store data as Linux files, and Dynamo is built on simple key/value paradigm by storing data as binary objects (i.e., blobs) identified by unique keys. Therefore, these

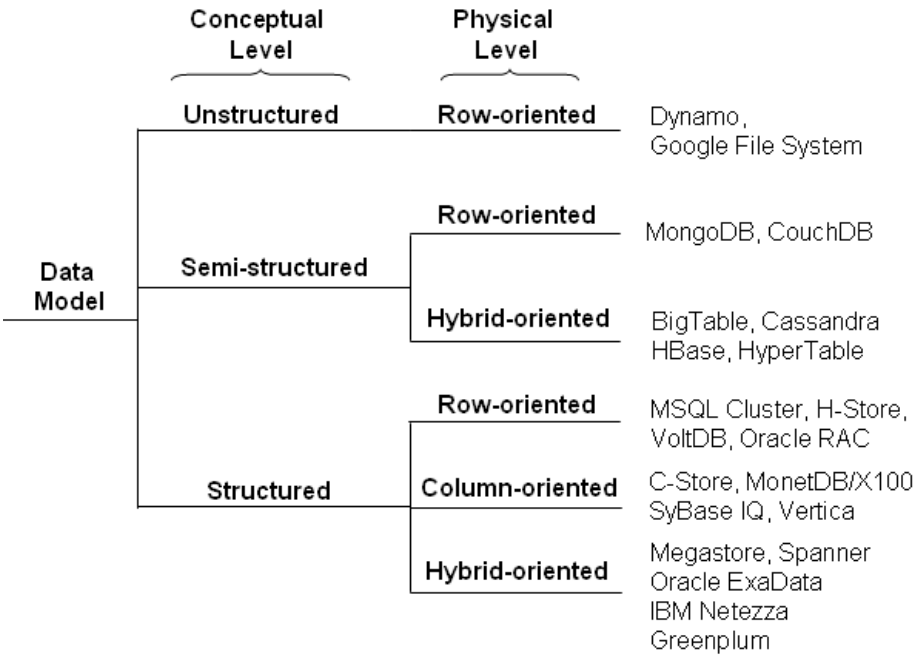

Fig. 1. Taxonomy of Data Model

two systems are both unstructured data stores, which can only use row-oriented physical layout.

Google's BigTable (Chang et al, 2010) together with Cassandra (Lakshman et al, 2010), HBase (HBase, 2009), HyperTable (HyperTable, 2008), which are BigTable-like systems, are representatives of semi-structured data store built on hybrid-oriented physical layout. They treat each individual table as a sparse, distributed, multi-dimensional sorted map, in which data is organized into three dimensions: rows, columns, and timestamps. This organization provides the semi-structured data. Column keys are grouped together into sets named column families that form the unit of access control. All data items stored in the same column family is usually stored continuously in the physical disk which is row-oriented layout. Client application can access values from individual rows and then iterate over subset of the data in a column family.

CouchDB (CouchDB, 2010) and MongoDB (MongoDB, 2009) considered as a document store (DocumentDB, 2012), are another typical class of semi-structured data store using row-oriented physical layout. Data is stored with the form of documents in XML, JSON (JavaScript Object Notation) or some other formats that can be embedded recursively inside. Since data in a document store is serialized from XML or JSON format, document stores usually adopt row-oriented layout similar to key/value stores.

C-store (Abadi et al, 2006; Stonebraker et al, 2005) supports the relational structured data model, whereas tables are stored column-oriented physically. MonetDB/X100 (Boncz et al, 2006; Manegold et al, 2009) and commercial systems Sybase IQ (Sybase IQ, 2010) and Vertica (Vertica, 2011) adopts a similar idea of C-store. These systems are typical relational databases with column-

oriented physical layout. Same as relational systems, columns or collections of columns can form unique primary key or be a foreign key referencing a primary key in other tables. Data stored in each column are sorted in the same order, thus values from different columns in same order position belong to the same logical row. These systems benefit greatly from data compression techniques. Having data from each column with same data type and low information entropy stored close together, the compression ratio can be dramatically enhanced saving a large amount of storage.

Megastore (Baker et al, 2011) and Spanner (Corbett et al, 2012) define a structured data model based on relational tables stored on BigTable. Since they are built on top of BigTable infrastructure, hybrid layout is applied on the physical level. The column name in the BigTable is a concatenation of the table name and the property name. Entities from different tables can be mapped into the same BigTable row. Each indexed entry is represented as a single BigTable row and the key of the cell is constructed using the indexed property values combined with the primary key of the indexed entity. Same as in traditional relational database, the data model is declared in a schema. Client applications can declare additional hierarchies in the database schemas. Tables are either entity group root tables or child tables, which must declare a single distinguished foreign key referencing a root table.

Oracle's Exadata (Oracle, 2012), IBM's Netezza Server (Francisco, 2011) and Greenplum (Cohen et al, 2009) evolved from traditional parallel database systems, and thus support structured data store. Furthermore, Exadata introduced Hybrid Columnar Compression (HCC) (Oracle, 2012) in the granularity of disk block. Netezza integrates row and column oriented layout on each individual table. Greenplum provides multiple storage mechanisms with a variety of format with different level of compression modes. Column store with slightly compressed format is applied for data that is updated frequently, and append-only tables are using row store with heavily compressed format. Therefore, these systems belong to hybrid-oriented layout category. Hybrid Columnar Compression employs the similar idea of Partition Attributes Across (PAX) (Ailamaki et al, 2001) combined with compression. In each disk block, records reside on the same block as row-oriented store, but values of the same column are stored sequentially together in disk blocks. This is different from the pure column-oriented store, which stores different columns in entirely separate disk blocks. HCC has the I/O characteristics of row store and cache characteristics of column store, providing the compression benefits of column-oriented layout without sacrificing efficient row-level access too much.

2.4. Data Model Scalability

??? In this subsection, we discuss the advantages and disadvantages of the various data models and conceptual levels, especially in terms of scalability.

To achieve high scalability, systems need to distribute data to different nodes. The simplest way for data distribution is to deploy individual tables at different nodes with no partition. However, since concurrent transactions usually access different portions of one table, thus data 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 (Cohen et al, 2009). Partitioning can be horizontal

or vertical, and systems require a close integration between data model and data partition. Each partition of a table can have a different storage format, to match the expected workload. For row-oriented data store, 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 data store, vertical or mixed partitioning can be applied. The fine-grained mixed partition schema is often adopted 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 categorized into static column groups (Grund et al, 2010; Jones et al, 2010). In addition, columns with large-size data are separated independently to take the advantage of the compression benefits in column store (Abadi et al, 2006). Column groups are further partitioned horizontally across multiple nodes.

Regarding to the conceptual level, numerous NoSQL large-scalable systems such as key-value stores (DeCandia et al, 2007) and BigTable-like stores (Chang et al, 2010; Lakshman et al, 2010) represent a recent evolution in building infrastructure by making trade-off between scalability and functionality. They adopt variant of the schema-less unstructured data for large-scale data application. However, some systems named as grid databases, or databases on the cloud, or NewSQL relatively to NoSQL, for data-centric applications, seek to provide the high scalability and throughput same as NoSQL while still preserving the high level query capabilities of SQL, and maintaining transactional guarantees (Kallman et al, 2008; Baker et al, 2011; Corbett et al, 2012; Cohen et al, 2009). The unstructured data model in lower conceptual level can simplify the design and implementation for scale-out capability, but data model in higher conceptual level is not an obstacle for scalability as long as an optimal partitioning strategy is applied based on the schema. Inappropriate partitioning may cause data skew. The skewed data will decline response time and generate hot nodes, which easily become a bottleneck throttling the overall performance and scalability (Kossmann et al, 2010; Xu et al, 2008). It has been observed that most OLTP workloads in reality can always be constructed as a tree structure. Every table in the schema, except the root one, has a many-to-one relationship to its ancestor (Das et al, 2010; Kallman et al, 2008). Tuples in every descendent table are horizontally partitioned according to the ancestor that they descend from, so that all data related to a client is mapped into the same node. As a consequence, data accessed by a transaction are usually deployed on the same node. The design of partitioning scheme is based on relational schema and query workload in order to minimize the contention.

??? Remove this paragraph. In this section, we discuss two key levels of data model for systems affecting the scalability and performance: physical layout and conceptual model. Different layouts and models are presented and compared by analyzing how different systems trade off and implement the appropriate integration according to various requirements.

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. The architecture is crucial and fundamental to achieve high availability, scalability, and performance. In this

section, we are going to classify large-scale systems according to diverse architectures. Database vendors and network enterprises have been endeavoring to implement systems providing considerable scalability from different dimensions to keep pace with growing big data application trends. There are four historical shifts in the architecture technology behind large-scale data management systems:

1. the invention of database on cluster of processors (single or multi-core) with shared memory.
2. the improvement of database on cluster of processors with distributed memory but common disk.
3. the rise of parallel data processing on shared-nothing infrastructure.
4. the popularization of MapReduce parallel paradigm and distributed file system.

3.1. SMP on Shared-Memory Architecture

In the symmetric multi-processing (SMP) on shared-memory infrastructure, a group of processors, each with its own cache, share a unique common memory and disk storage. Threads of codes are automatically allocated to processors for execution, and workload is balanced among the processors by the operating system. Resources such as memory and the I/O systems are shared and accessed equally by each of the processor, as illustrated by Figure 2. This architecture involves a pool of tightly coupled homogeneous processors running separate programs and working on different data with the capability of 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 major commercial database vendors have provided products based on it. However, the architecture has the underlying deficiencies in scalability for processing large amounts of data.

??? (1) citation, and (2) any upper limit for such architecture.

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 3. Each processor within an SMP cluster node shares 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) (Gibson and Van Meter, 2000). For instance, Oracle and HP grid solution orchestrates multiple small server nodes and storage subsystems into one virtual machine based on SAN (Poess and Nambiar, 2005). Unlike shared-memory infrastructure, there is no common memory location to coordinate the sharing of the data. Hence explicit coordination protocols such as cache coherency (Bridge et al, 1997) and cache fusion (Lahiri et al, 2001) are needed. ??? citation

The MPP architecture on the hybrid combination of SMP clusters is commonly used in several well-known scalable database solutions. Two notable sys-

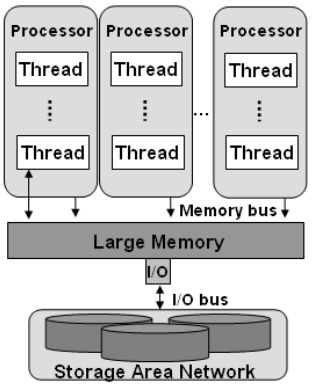

Fig. 2. SMP on Shared-Memory Architecture

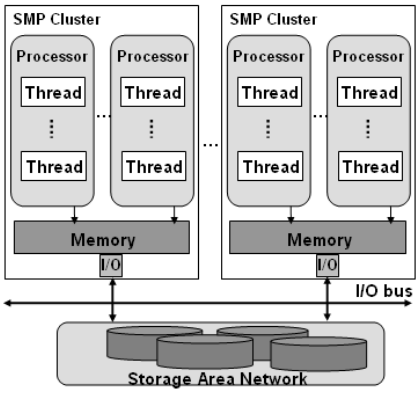

Fig. 3. MPP on Shared-Disk Architecture

tems using this architecture are Oracle Exadata Database Machine (Oracle, 2012) and IBM Netezza Performance Server (NPS) (Francisco, 2011). Exadata is a complete, pre-configured Oracle system that combines Oracle RAC (Oracle RAC, 2012) with new Exadata Storage Servers. Exadata improves parallel I/O and filters only data of interest before transmitting. Netezza integrates server, storage and database all in a single compact platform. It proposed Asymmetric Massively Parallel Processing (AMPP) mechanism with Query Streaming technology that is an optimization on hardware level (Francisco, 2011).

3.3. Sharding on Shared-nothing Architecture

Based on the idea that data management system can be parallelized to leverage multiple commodity processors in a network to deliver increased scalability and performance, the parallelism on shared-nothing infrastructure was coined for these new computing clusters, to distinguish the architecture on the shared-resource infrastructure. The sharding on shared-nothing infrastructure is currently a most widely used architecture for large-scale data (Ghemawat et al, 2003; Chang et al, 2010; DeCandia et al, 2007; Baker et al, 2011; Cooper et al, 2008; Lakshman et al, 2010; Campbell et al, 2010; Ronstrom and Thalmann, 2004).

In order to harness the power of this architecture, data is partitioned across multiple computation nodes, which results in dramatic performance improvements with parallel query execution. The purpose for this is to achieve the scalability and parallelism advantages of shared-nothing infrastructure while reducing costs and minimizing administration overheads. Each machine hosts its own independent instance of database system with operating on its portion of the data. Each machine is highly autonomous, performing its own scheduling, storage management, transaction management, concurrency control and replication. Sharding on shared-nothing architecture is a two-tiered system design, as shown in Figure 4. The lower processing unit tier is composed of dozens to hundreds or thousands 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

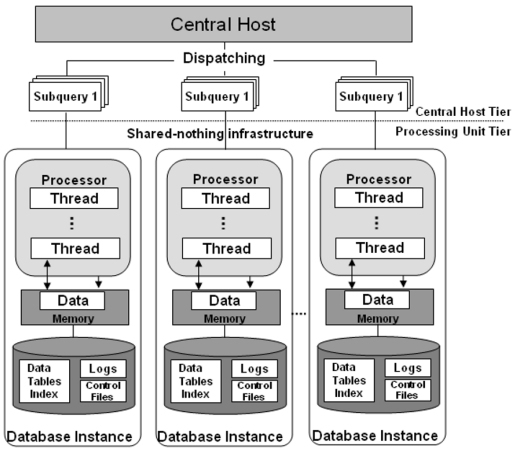

Fig. 4. Sharding on Shared-Nothing Architecture

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.

This approach facilitates all operations to be easily parallel processed on each unit, including parsing, records access, interlocking and logging, significantly reducing the contention of MPP on shared-disk approach. The autonomy of processing unit also creates excellent opportunity for a big data system, allowing additional units to be involved without concern about interruption with other units. This architecture allows high scalability, as long as the scope of transaction is limited into a single node, that is, each DBMS instance is hosted on one node and processes queries without contacting with any other instance. There are two flavors of this architecture that are *centralized topology* and *decentralized topology*.

Centralized Topology. Centralized topology utilizes a logically centralized coordinator to manage 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 for the key, routing the request to the nodes and integrating for the responses. The centralized approach simplifies the design and implementation of the complex architecture since the central node has an authoritative view of the whole system (Cooper et al, 2008; Ronstrom and Thalmann, 2004).

To prevent the central master server from becoming the bottleneck due to the heavy workload, shadow master mechanism is employed (Ghemawat et al, 2003). The key idea is to separate the control flow and data flow in 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. Clients only interact with the master for metadata information such as which data server should contact, and caches the information for a limited time. Particularly, GFS separates file system control, which passes through the master, from data transfer, which passes directly between chunk-servers and

clients. Master involvement in common operations is minimized by a large chunk size and by chunk leases, which delegate authority to primary replicas in data mutations. Besides prevent centralized master from becoming a bottleneck, this design also delivers high aggregate throughput for high concurrent readers and writers performing a variety of task.

Decentralized Topology. Unlike the centralized topology, systems such as Dynamo (DeCandia et al, 2007) and Cassandra (Lakshman et al, 2010) 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. Since there is no explicit central node holding membership information, any node status modification must be propagated via a gossip-based protocol (Birman, 2007) to other nodes of the system. The gossip-based membership protocol is a classical mechanism to exchange status among a large number of nodes efficiently in a scalable manner. The gossip protocol makes sure that every node keeps a routing table locally and is aware of the up-to-date state of other nodes. All servers are organized in a ring of successor nodes structure. Consistent hashing (Karger et al, 1997) is widely used in the decentralized 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 a request to any random node, and the node will forward the request 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 such as search engine infrastructure and messaging. Interesting architectures have been proposed to deal with massive concurrent requests on large data volume for excessive user basis. One representative design is the well-known MapReduce Framework, proposed by Google for processing large data sets, in which the Map and Reduce modules utilize partition parallelism to enable many Map and Reduce tasks to run in parallel (Dean and Ghemawat, 2008). 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 demand exceeds service capacity (Welsh et al, 2001).

Google developed MapReduce, which is highly scalable and parallel for big data processing (Dean and Ghemawat, 2008). Applications programmed with this framework are automatically parallelized and executed on a large cluster of commodity machines. The run-time system schedules the programs execution across a set of machines and manages the required inter-machine communication. The framework consists of two abstract functions, Map and Reduce, which can be considered as two different stages. The Map stage reads the input data and produces a collection of intermediate results; the following Reduce stage pulls the output from Map stage, processes to final results.

MapReduce can program over high performance database tables. Since database tables are partitioned across multiple nodes, the initial Map phase is executed in the local database engine directly on the partition, providing fully parallel I/O with computation "pushed" to the data. A stand alone MapReduce engine

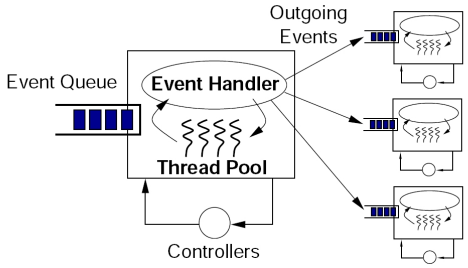

Fig. 5. Staged Event Driven Architecture

requires specified data access routines in their MapReduce script, which would then access a remote database server via a connectivity protocol, and then "pull" the records over to Reduce phases (Cohen et al, 2009).

Staged Event-Driven Architecture is designed based on event-driven approach that has been introduced and studied for various software applications, such as Dynamic Internet Servers and high performance DBMSs (Harizopoulos and Ailamaki, 2003; Welsh et al, 2001). 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 5 (Welsh et al, 2001). SEDA breaks the execution of application into a series of stages connected by explicitly associated queues. Each stage represents sets of states from the FSM in the event-driven design, and can be regarded as an independent, self-contained entity with its own incoming event queue. Stages pull a sequence of requests, one 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 queues of the next stage. Each stage is isolated from one another for the purposes of easy resource management, and queues between stages decouple the execution of stages by introducing explicit control boundaries. The SEDA design makes use of the dynamic resource controllers to monitor the load by an associated stage (Welsh et al, 2001).

The SEDA design has been applied to improve the database performance through exploiting and optimizing locality at all levels of memory hierarchy of single symmetric multiprocessing system at the hardware level (Harizopoulos and Ailamaki, 2003). The SEDA provides a satisfied design for high scalable system, since stages can be easily arranged to run on various node servers. As a matter of fact, it has been shown that the aforementioned MapReduce framework can also regard as an architecture based on SEDA, and the basic MapReduce framework resembles the two-stage architecture. ??? citation

3.5. System Architecture Taxonomy

Based on the above analysis, we present the taxonomy of large-scale data management system architecture in Figure 6.

Due to the long-time popularity of shared-memory multi-threads parallelism, almost all major traditional commercial DBMS providers support products with

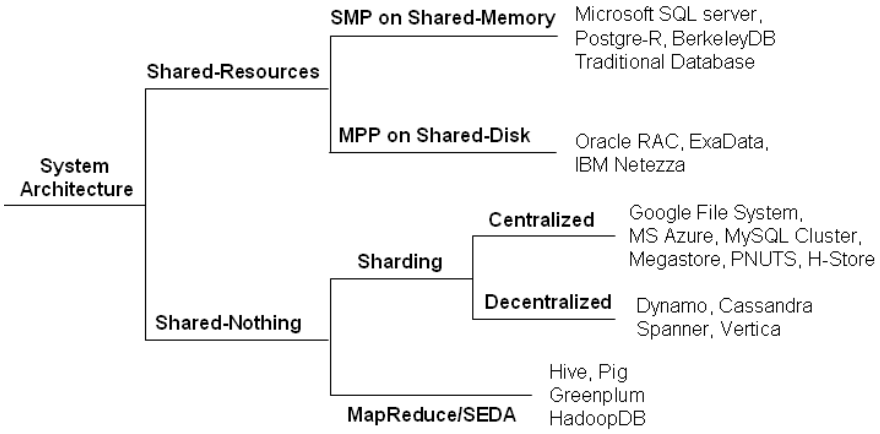

Fig. 6. Taxonomy of system architecture

SMP on shared-memory architecture, such as Microsoft SQL Server, Oracle Berkeley DB and Postgres-R, to name a few.

??? This whole paragraph seems off the topic. The large-scale data exacerbates I/O issue, thus one trend is developing highly parallelized I/O between storage and central processing units with MPP on shared-disk architecture. Representative systems of this trend include Oracle RAC (Oracle RAC, 2012), Exadata (Oracle, 2012) and IBM Netezza (Francisco, 2011). The high scalability and performance is delivered for large-scale data applications by leveraging a massively parallel fashion within a collection of storage cells. These systems push as much SQL work as it can to storage cells. Each storage cell returns only the requested rows and columns that fulfill the requirements of a query rather than querying the entire table itself. 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 performance. Systems such as Exadata also adopt compression technique. Only the data being returned to the clients are decompressed, which can reduce the I/O well. These database systems can provide high performance for processing large-scale data relying on dedicated hardware and software implementation, where dedicated high-end hardware combined with complex software protocols is applied for the fundamental infrastructure.

Microsoft Azure server (Campbell et al, 2010) 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 much of the relational features of 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 for the internal database structures in the server.

MySQL Cluster (Ronstrom and Thalmann, 2004) is a typical sharding on shared-nothing, distributed node architecture storage solution designed for high scalability and performance 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 management nodes.

H-Store (Kallman et al, 2008) is a highly distributed, row-store oriented relational database that runs on a cluster on shared-nothing infrastructure, main memory executor nodes. H-Store provides an administrator within the cluster that takes a set of compiled stored procedures as input.

Google File System (GFS) (Ghemawat et al, 2003) is a scalable distributed file system for large distributed data-intensive applications. A GFS cluster uses a simple design with selecting a single master server for hosting the entire metadata and the data is split into chunks and stored in chunk-servers hosted on other nodes.

BigTable (Chang et al, 2010) is built on GFS, and relies on a highly-available and persistent distributed lock service for master election and location bootstrapping. To avoid the master from being a bottleneck, aggressively caches are applied to minimize data transmit and save the processor time and bandwidth.

Megastore (Baker et al, 2011) is a higher layer over BigTable. 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. To improve query efficiency, both local indexes in each entity group and global indexes across multiple entity groups are realized.

Yahoo! PNUTS (Cooper et al, 2008) 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.

GFS, BigTable, Megastore and PNUTS all elect and utilize logically central nodes to manage the coordination of the whole cluster, thus they all belong to the centralized topology category of sharding on shared-nothing architecture. Some systems such as Dynamo (DeCandia et al, 2007), Cassandra (Lakshman et al, 2010) and Spanner (Corbett et al, 2012) opt symmetric structure on decentralized topology over centralized one based upon the understanding that symmetry in decentralization can simplify the system provisioning and maintenance. Systems with decentralized topology basically employ a distributed agreement and group membership protocol to coordinate actions between nodes in the cluster. The messaging protocol uses broadcast and point-to-point delivery to ensure that any control message is successfully received by other nodes.

Dynamo used the techniques originate in the distributed systems research of the past years such as DHTs (Gummadi et al, 2003), consistent hashing (Karger et al, 1997), vector clocks (Lamport, 1978), quorum (Alvisi, 2001). Dynamo is the first production system to use the synthesis of all these techniques (DeCandia et al, 2007).

Facebook Cassandra (Lakshman et al, 2010) 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 Google's BigTable and the distributed systems technologies from Dynamo.

Spanner (Corbett et al, 2012) is a scalable, multi-version, globally-distributed database based on True Time API, which combines an atomic clock and a GPS clock to timestamp data so it can then be synchronized across multiple machines without the need for centralized control.

Hive (Thusoo et al, 2009; Thusoo et al, 2010), an open-source data warehousing solution built on top of Hadoop, compiles SQL-like declarative queries into a directed acyclic graph of MapReduce jobs executed on Hadoop. Pig (Olston et al, 2008) is a high-level data flow system, in which programs are compiled into sequences of Map-Reduce tasks and executed on the Hadoop cluster. Pig first translates a execution plan into a physical plan, and then assign each physical operator inside a MapReduce stage to construct a MapReduce plan.

HadoopDB (Abouzeid et al, 2009) 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. The above systems relies on MapReduce framework to ensure the scalability of Hadoop.

We have presented four different system architectures: symmetric multiple processing on shared-memory, massively parallel processing on shared-disk, sharding on shared-nothing and MapReduce/SEDA. These architectures are supported by diverse database providers. Now we will discuss the scalability of these architectures.

3.6. Architecture Scalability

??? It is better to just present a simpler conclusion that in terms of scalability, we have MapReduce/Staged >> Sharding Decentralized > Sharding Centralized >> Shared MPP > Shared SMP, with some numbers and/or citations.

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 resource limitation. In particular, dozens of processors are sharing memory and storage. When data volumes are increasing enormously, systems based on SMP will run out of their memory. As processors turns to access the large amount of data in memory, the memory bus becomes a bottleneck easily. When the bus is saturated, increasing the number of processors or the size of memory is ineffective. Hence, memory bus bandwidth is one limit ceiling for scaling-up. Similarly, I/O bus bandwidth can also be clogged with traffic as the amount of data transferred increases, adversely affecting the performance.

In addition, multi-processors require sophisticated logical mechanism in the hardware to keep the L2-cache consistent, so it is rather difficult for a single machine to have hundreds of processors unless they are specially customized from non-commodity devices and if so, it will be very expensive but rather cost-efficient. The SMP on shared-memory systems have the disadvantage of limited scalability leading to a progressive decline in performance as the amount of big data grows.

Similarly, in the systems based on shared-disk infrastructure, the internal bus and I/O connections cannot handle the tremendous amount of traffic, which create a performance bottleneck that inhibits scalability. In addition to bus bandwidth, threads contention is another factor hindering the scalability as the hardware contexts grow exponentially. Multi-threading is the main mechanism for utilizing shared resources. Each incoming query is handled by one thread. Each thread executes until it either blocks on a synchronization condition such as I/O wait, or until a predetermined time quantum has elapsed. Then the CPU switches context and executes a different thread or a different task. The overhead caused by the context switches can be considerable high (Stonebraker et al, 2007). In

addition, complex locking manipulation is essential to control threads to access critical data section. With excessive threads, some critical section will eventually become a bottleneck, since critical sections need to serialize the threads which compete for them (Johnson et al, 2009).

Due to the fact that data in the shared-disk needs to be dynamically cached in multiple processors memory to exploit access locality, synchronization of reads and writes requires locks management, and invalidations of stale cached data or propagations of updated data must be conformed. The overheads induced by the locking mechanism will become high as the system is scaled up, which will hinder the system performance sharply. To sum up, the systems fundamental dependency on a shared-resource design limits its ability to scale for the massively concurrent requirements.

In the shared-nothing infrastructure, local resources serve local processor; thus it overcomes the disadvantage of limited bus bandwidth and limited memory size. Furthermore, it can be better for communication with lower cost. The shared-nothing design is intended to support smooth extensibility of the system by involving new computation nodes. Therefore, sharding and SEDA on the shared-nothing architecture are desired ways to achieve ultimate scalability, and a large number of applications based on MapReduce framework are such good examples. Google uses it internally to process more than 20PB data set perday. MapReduce reaches the ability to sort 1 PB data using 4,000 commodity servers (Dean and Ghemawat, 2008). Hadoop at Yahoo! is assembled on 3,000 nodes with 16PB raw disk, 64TB of RAM and 32K CPU cores (Shvachko et al, 2010). At Facebook, Hive (Thusoo et al, 2009) forms the storage and analytics system infrastructure that stores 15PB data and process 60TB new data everyday (Thusoo et al, 2010).

??? Again, this paragraph is not properly located. In this section, we discuss four different system architecture on top of shared-resources and shared-nothing infrastructure. Architectures on shared-resources have limited scalability due to the inherent resource contention. Sharding and SEDA on shared-nothing infrastructure are ideal architectures to achieve high scalability.

4. Consistency Model

One of the challenges in the design and implementation of a big data management system 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 (Lewis et al, 2002), are often renounced in favor of weaker ones, such as *BASE* (Cooper et al, 2008). In this section, we are going to classify systems according to different consistency level 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*) (Lewis et al, 2002). Database management systems with ACID properties

provide different isolation levels, include *read committed*, *Snapshot isolation*, and *serializability* (Berenson et al, 1995).

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 multiversion 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 allows snapshot isolation to avoid the common type of lost update anomaly (Bornea et al, 2011).

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

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 updates 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* (Bornea et al, 2011), in which the effects of execution a set of transactions are equivalent to executing the transactions in the serial order within only 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 (Ronstrom and Thalmann, 2004; Lin et al, 2005).

4.2. BASE Properties

The ACID properties work fine for 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 paradigms are favored and the network partition or node failure can be normal rather than an exception. Naturally, Many large-scale distributed systems, such as Amazon Dynamo (DeCandia et al, 2007), Yahoo! PNUTS (Cooper et al, 2008) and Facebook Cassandra (Lakshman et al, 2010), 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) (Pritchett, 2008).

Various BASE consistency models have been specified, and thus we first categorize these models and present examples to demonstrate different levels of consistency guarantees.

4.2.1. Eventual Consistency

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

Even though a system with eventual consistency guarantees to to "eventually" converge to a consistency state, it increases the complexity of distributed software applications because eventual consistency does not make safety guarantees: an eventually consistent system can return any value before it converges (Pritchett, 2008; Lloyd et al, 2011).

However, 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.

4.2.2. Causal Consistency

Causality informally 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 (Ahamad et al, 1995).

A write-into order $\mapsto$ is a relation with the following properties:

a) If $op_1 \mapsto op_2$, then there is a data object such that op_1 writes it earlier, and op_2 reads it later.
b) For any operation op_2, there is at most one op_1 such that $op_1 \mapsto op_2$.
c) If op_2 reads some data object and there is no op_1 such that $op_1 \mapsto op_2$, then op_2 return $\perp$.

The write-into order defines the relative ordering of related 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. Causal consistency is the transitive closure of the union of the write-into order and the preceding order. In the typical key-value stores, the causal consistency guarantees the causal order relation $\prec$ for cases as below (DeCandia et al, 2007; Lloyd et al, 2011):

a) Two operations a and b are submitted on the same node and $a \mapsto b$, then $a \prec b$.
b) Two operations a and b are submitted on different nodes. If a has been received and executed before b is submitted, then $a \prec b$.
c) If $\exists c, c \neq a, c \neq b$ such that $a \prec c$ and $c \prec b$, then $a \prec b$. This property defines the transitive closure of the first two orders.

Operations without causal dependency are commutative. If two consecutive operations commute, they can execute in either order. Implementation of causal consistency usually involves dependency tracking (Lloyd et al, 2011; Bailis et al, 2013). Dependency tracking associated with each operation is employed to record meta-information for reasoning about causality order. Each process server reads from their local data objects 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 dependent operations, ordering consistency is an enhanced variation of causality consistency ensuring global ordering of operations. Ordering consistency provides the monotonicity guarantee of both read and write operations to each record.

(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 (Saito and Shapiro, 2005). It can be achieved by designating one node as the primary node for every record; and then all updates to that record are 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 to all relevant nodes. In the case that the primary node fails, one of the non-primary nodes is elected to act as the new primary node. To guarantee successive reads by the same user return increasingly up-to-date results, the node the user access is always the same one.

From the analysis above, it is not difficult to see that considering consistency is stronger than causal consistency in that a system that guarantees ordering consistency also guarantees causal consistency, but not vise versa. Further, causal consistency is stronger than eventual consistency.

If we use $\succ$ to represent the stronger relationship among two consistency models, the following demonstrates that all the three consistency models form a linear order: (??? citation or formal proof)

$$\textit{ordering consistency} \succ \textit{causal consistency} \succ \textit{eventual consistency}$$

??? NOTE: We may also to discuss the relationships among ACID and BASE: that is

$$\textit{serializability} \succ \textit{snapshot} \succ \textit{read committed} \succ \textit{ordering consistency}$$

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 7. The classification is based on our ensuing analysis.

Spanner (Corbett et al, 2012), Megastore (Baker et al, 2011) and Spinnaker (Rao et al, 2011) provide *one-copy serializability* with a Paxos-based protocol (Chandra et al, 2007). Paxos ensures that data will be available as long as the majority of the nodes are alive. Megastore and Spinnaker implement with two phases: a leader election phase, followed by a quorum phase where the leader proposes a write and followers accept it. Spanner's Paxos implementation uses TrueTime leases, in which Spanner assigns a TrueTime to each write operation in monotonically increasing order, allowing Spanner to correctly determine whether a state is sufficiently up-to-date to satisfy a read.

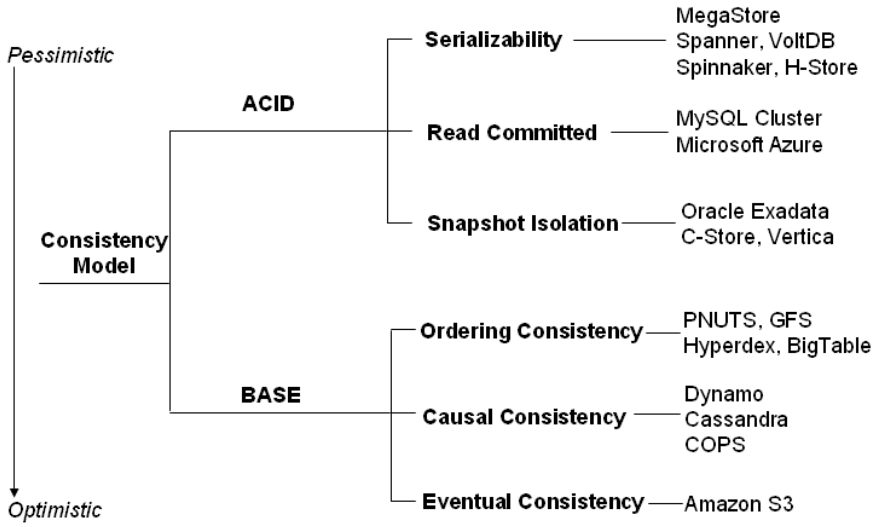

Fig. 7. Taxonomy of Consistency Model

VoltDB and H-Store (Kallman et al, 2008; Jones et al, 2010; Stonebraker et al, 2007) support SQL transaction execution through stored procedure. By initiating 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 such systems resemble the single operation call.

MySQL Cluster (Ronstrom and Thalmann, 2004) and Azure (Campbell et al, 2010) combined traditional read committed with the master-slave mode synchronization. Exadata (Oracle, 2012) isolates read-only transactions using snapshot isolation. High water mark with low-overhead mechanism is introduced for keeping track of its value in multi-replica environment in C-Store (Stonebraker et al, 2005).

Hyperdex (Escriva et al, 2012) provide ordering consistency with chaining, 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; Otherwise, read operations will be forwarded to the tail node.

Yahoo! PNUTS (Cooper et al, 2008) provides a per-record time line consistent model which 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 the identical committed sequence. Update is considered to be committed when it is published to the broker. After being committed, the update will be asynchronously propagated and applied to different nodes. The update will be buffered by the broker until it is applied to all nodes.

GFS (Ghemawat et al, 2003) and BigTable (Chang et al, 2010) both use Chubby (Burrows, 2006), a distributed locking mechanism for distributed node coordination. The GFS relies 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. In the BigTable, Chubby provides a namespace consisting of directories and small files. Each directory or file can be added a lock, and reads and writes to a file are atomic.

Dynamo (DeCandia et al, 2007), Voldemort (Voldemort, 2011) and Cassandra (Lakshman et al, 2010) provide causal consistency for allowing applications with "always writeable" property. In Dynamo, write operations can be accepted by any node. Vector clock, also named version vector or timestamp vector, is associated with multiversion of data to determine the causal dependency order during reconciliation. Dynamo and Cassandra resolve conflicts during read operations. Some other systems, such as COPS (Lloyd et al, 2011; Burckhardt et al, 2010), enforce dependency resolution on write operations to achieve "nonblock read" property.

4.4. Consistency Scalability

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 tradeoff between consistency and scalability, and especially how the implementations of different consistency models affect the scalability.

The common implementation to provide serializability is based on distributed two-phase locking (2PL) protocol. However, 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 (Larson et al, 2011; Özsu, 2011; Weikum and Vossen, 2001). 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 (Özsu, 2011). The limitation of this implementation is closely related to the problem of testing a distributed graph for cycles, which also arises the deadlock detection. Thus, transactions executed by a distributed database should not cause distributed deadlocks because such deadlocks are difficult and expensive to detect. Optimistic protocols are lock-free assuming that conflicts between transactions are rare. However, to guarantee that validation phase can produce consistent results, a global order checking is required which will decrease performance heavily for large-scale distributed data system (Corbett et al, 2012). In the presence of slower network connection, more transactions may crowed into the system causing excessively high chances of rollbacks (Larson et al, 2011).

The communication latency caused by various commit protocol implementations (e.g. two-phase commit, three-phase commit, Paxos, etc.) can limit the scalability as well. A protocol implementation performs well within local-area networks where latency can almost be negligible and failures are rare. However, it may severely limit the scalability of large systems with frequent update operations since the availability and coordination overhead become worse as the number of nodes increases (Yu and Vahdat, 2002; Yu and Vahdat, 2006). For example, the well-known two-phase commit protocol, commonly used in traditional distributed database systems, scales poorly due to the overhead caused by mul-

tiple network round-trips (Helland, 2007). Constraining the scope of transaction is a way to minimize communication overhead. Azure (Campbell et al, 2010), Megastore (Baker et al, 2011) and H-Store (Kallman et al, 2008) only support single-node transaction that can be executed to completion in a single node, and one-shot transaction (Stonebraker et al, 2007) that can be executed in parallel with no communication among nodes. This restrictive scope is only reasonable in the applications where data can be well deployed, since distributed transactions will be very rare in such cases. Despite of the high scalability, this alternative implementation can hinder the concurrency and applicability of database fro generalized workload.

BASE can achieve high scalability much easier than ACID. Though eventual consistency can achieve linear scalability, it has its own potential disadvantages. First, *eventual consistency* makes only liveness rather than safety guarantee, as it merely ensures the system to converge to a consistent state in the future. Second, 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 (Cooper et al, 2008; DeCandia et al, 2007; Shute et al, 2013). Scaling the causal consistency may lead to a trade-off between throughput and latency (Bailis et al, 2012). Particularly, to preserve the causality, a new write must be waiting for its dependencies before it can be applied. For this purpose, each node maintains a data structure representing the complete graph of dependency. The amount of time that the update is invisible is determined by both network latency and the complexity of dependency graph. Though vertices and edges can be removed from the dependency graph eventually, the graph must be preserved for the whole lifetime of the operation. As a result, in the presence of heavily concurrent workloads, the dependency graph can potentially expand to an extremely complex structure, limiting capacity and throughput. Systems with causal consistency always provide only single-record basis operation such that the probability that different operations access the same data object is greatly reduced, and thus the dependency graph is simplified so that the scalability will not be limited by it. The total ordering protocol ensures that operations are received in the identical order at all sites. Each operation is initiated by sending read and write pre-declares to corresponding global scheduler node as a single atomic action in totally ordered fashion. Though the implementation of the total ordering algorithm performs efficiently with low network latency, it appears to be a rather disadvantage approach for high network latencies (Cooper et al, 2008).

??? The paragraph, again, seems not in a proper place. In this section, we explore analysis on manipulating different consistency models in large-scale data management systems. The critical problem is to cope with the subtle and difficult issues of keeping data consistency in the presence of highly concurrent data accesses. Developing a suitable consistency model is an important aspect for big data systems.

5. Conclusion

In this survey, we have investigated, studied, characterized, and categorized several critical aspects of large-scale data management systems. These systems have several unique characteristics including scalability, elasticity, manageability, and low cost-efficiency. We have enumerated various data storage models about data physical layout and data conceptual representation. Further on, we focused on

the design and implementation of system architecture. We have developed architecture taxonomies for prevailing large-scale database systems to classify the common architecture designs and to provide a basis for analysis of the scalability limitations. These limitations represent some of the directions that can be taken for future exploration.

We then compared two categories of the consistency models and classified prevailing systems to the respective taxonomies. With this mapping, we have gained an insight into the strategies and practices which current systems are issued with consistency and reliability at their core. Throughout our characterization, we can identify some of the limitations and discover vagueness of comparability. For the purpose of verifying diverse protocol implementations, we analyze and compare different consistency models from several orthogonal dimensions. More suitable consistency models and effective protocols are still desired for large-scale data management systems.

To conclude, an in-depth understanding and a precise classification are essential for analyzing large-scale data management system and ensuring a smooth transition from conventional enterprise infrastructure to the next generation of large-scale infrastructure for big data applications. This survey delves deeper to lay down a comprehensive taxonomy framework that, not only servers as a direction for understanding the big data systems, but also presents a reference for which future efforts need to be undertaken by researchers.

References

Abadi D, Madden S, Ferreira M (2006) Integrating compression and execution in column-oriented database systems. In Proceedings of the 2006 ACM SIGMOD international conference on Management of data, Chicago, IL, USA, 2006, pp 671–682

Abadi DJ, Madden SR, Hachem N (2008) Column-Stores vs. Row-Stores: How different are they really?. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data 2005, June. pp 967–980

Abouzeid A, Bajda-Pawlikowski K, Abadi D, Silberschatz A, Rasin A (2009) HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In Proceedings of the VLDB Endowment, 2009, Aug. 2(1) 922–933

Agrawal D, Das S, Abbadi A (2011) Big data and cloud computing: current state and future opportunities. In Proceedings of the 14th International Conference on Extending Database Technology, Uppsala, Sweden, 2011, pp 530–533

Ahamad M, Neiger G, Burns J, Kohli P, Hutto P (1995) Causal memory: definitions, implementation, and programming. In Distributed Computing, 1995, 9(1) 37–49

Ailamaki A, DeWitt DJ, Hill MD, Skounakis M (2001) Weaving Relations for Cache Performance. In Proceedings of the 27th International Conference on Very Large Data Bases, 2001 pp 169–180

Alvisi L, Malkhi D, Pierce E, Reiter MK (2001) Fault Detection for Byzantine Quorum Systems. IEEE Transaction Parallel Distribute System. 2001 Sep., 12(9). 996–1007

Bailis P, Fekete A, Ghodsi A, Hellerstein JM, Stoica I (2012) The potential dangers of causal consistency and an explicit solution. In Proceedings of the Third ACM Symposium on Cloud Computing, 2012, 1(22) 1–7

Bailis P, Ghodsi A, Hellerstein JM, Stoica I (2013) Bolt-on causal consistency. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, 2013, pp 761–772

Baker J, Bond C, Corbett J, Furman JJ, Khorlin A, Larson J, Léon JM, Li Y, Lloyd A, Yushprakh V (2011) Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In the 5th Conference on Innovative Data Systems Research, 2011 Jan., pp 223–234

Berenson H, Bernstein P, Gray J, Melton J, O'Neil E, O'Neil P (1995) A critique of ANSI SQL

isolation levels. In In Proceedings of the 1995 ACM SIGMOD international conference on Management of data, 1995, pp 1–10

Birman K (2007) The promise, and limitations, of gossip protocols. In SIGOPS Operation System Review, 2007, Oct. pp 8–13

Boncz P, Grust T, Keulen M, Manegold S, Rittinger J, Teubner J (2005) MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In Proceedings of the 2006 ACM SIGMOD international conference on Management of data, Chicago, IL, USA, 2006, pp 479–490

Bornea M, Hodson O, Elnikety S, Fekete A (2011) One-copy serializability with snapshot isolation under the hood. In Proceedings of the 2011 IEEE 27th International Conference on Data Engineering, 2011, pp 625–636

Brantner M, Florescu D, Graf D, Kossmann D, Kraska T (2008) Building a database on S3. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, Vancouver, Canada, 2008, pp 251–264

Bridge W, Joshi A, Keihl M, Lahiri T, Loaiza J, MacNaughton N (1997) The Oracle Universal Server Buffer. In Proceedings of the 23rd international conference on Very Large Data Bases, 1997 pp 590–594

Burckhardt S, Leijen D, Fähndrich M, Sagiv M (2010) Eventually consistent transactions. In Proceedings of the 21st European conference on Programming Languages and Systems, 2010, pp 67–86

Burrows M (2006) The Chubby lock service for loosely-coupled distributed systems. In Proceedings of the 7th symposium on Operating systems design and implementation, Seattle, Washington, 2006, pp 335–350

Campbell DG, Kakivaya G, Ellis N (2010) Extreme Scale with Full SQL Language Support in Microsoft SQL Azure. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, 2010, pp 1021–1024

Chaiken R, Jenkins B, Larson P, Ramsey B, Shakib D, Weaver S, Zhou J (2008) SCOPE: easy and efficient parallel processing of massive data sets. In Proceedings of the VLDB Endowment, 2008 Aug., 1(2) 1265–1276

Chandra TD, Griesemer R, Redstone J (2007) Paxos made live: an engineering perspective. In Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing, 2007, pp 398–407

Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE (2006) Bigtable: A Distributed Storage System for Structured Data. ACM Transactions on Computer Systems (TOCS) 26(2), 4:1–4:26

Cohen J, Dolan B, Dunlap M, Hellerstein JM, Welton C (2009) MAD skills: new analysis practices for big data. In Proceedings of the VLDB Endowment, 2009, Aug. 2(2) 1481-1492

Condie T, Conway N, Alvaro P, Hellerstein JM, Gerth J, Talbot J, Elmeleegy K, Sears R (2010) Online Aggregation and Continuous Query Support in MapReduce. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, 2010, pp 1115–1118

Cooper BF, Ramakrishnan R, Srivastava U, Silberstein A, Bohannon P, Jacobsen HA, etc. (2008) PNUTS: Yahoo!'s hosted data serving platform. In Proceedings of the VLDB Endowment, 2008. 1(2) 1277–1288.

Corbett JC, Dean J, Epstein M, Fikes A, Frost C, Furman JJ, Ghemawat S, Gubarev A, Heiser C, Hochschild P, Hsieh W, etc. (2012) Spanner: Google's globally-distributed database. In Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation, Hollywood, CA, USA 2012, pp 251–264

CouchDB Website (2010) http://couchdb.apache.org/. 2010

Das S, Agrawal D, El Abbadi A (2010) G-store: a scalable data store for transactional multi key access in the cloud. In Proceedings of the 1st ACM symposium on Cloud computing, 2010. pp 163--174.

Dean J, Ghemawat S (2008) MapReduce: Simplified Data Processing on Large Clusters. Communication ACM 51(1) 107--113

DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W (2007) Dynamo: Amazon's Highly Available Key-Value Store. In Proceedings of 21st ACM SIGOPS symposium on Operating systems principles, Stevenson, Washington, USA, 2007, pp 205--220

DocumentDB Website (2012) http://en.wikipedia.org/wiki/Document-oriented_database. 2012

Escriva R, Wong B, Sirer E (2012) HyperDex: a distributed, searchable key-value store. In SIGCOMM Comput. Commun. Rev., 2012, Oct. 42(4) 25--36

Francisco P (2011) The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. IBM Red Paper. IBM Co., Ltd. 2011

Ghemawat S, Gobioff H, Leung ST (2003) The Google file system. In Proceedings of the 19th ACM symposium on Operating systems principles, Bolton Landing, NY, USA, 2003, pp 29--43

Gibson GA, Van Meter R (2000) Network attached storage architecture. In Communications of the ACM, 2000. 43(11) 37--45

Grund M, Krüger J, Plattner H, Zeier A, Cudre-Mauroux P, Madden S (2010) HYRISE: a main memory hybrid storage engine. In Proceedings of the VLDB Endowment, 2010, Nov. 4(2) 105--116

Gummadi K, Gummadi R, Gribble S, Ratnasamy S, Shenker S, Stoica I (2003) The impact of DHT routing geometry on resilience and proximity. In Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, Karlsruhe, Germany, 2003, pp 381--394

Guo H, Larson P, Ramakrishnan R, Goldstein J (2004) Relaxed currency and consistency: how to say "good enough" in SQL. In Proceedings of the 2004 ACM SIGMOD international conference on Management of data, 2004, pp 815--826

Hankins RA, Patel JM (2003) Data morphing: an adaptive, cache-conscious storage technique. In Proceedings of the 29th international conference on Very large data bases, Berlin, Germany, 2003, pp 417--428

Harizopoulos S, Ailamaki A (2003) A Case for Staged Database Systems. In CIDR, 2003

HBase Website (2009) http://hbase.apache.org/. 2009

Helland P (2007) Life beyond distributed transactions: an apostate's opinion. In 3rd Biennial Conference on Innovative Data Systems Research, 2007, pp 132--141

HyperTable Website (2008) http://hypertable.org/. 2008

Johnson R, Pandis I, Hardavellas N, Ailamaki A, Falsafi B (2009) Shore-MT: a scalable storage manager for the multicore era. In Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, Saint Petersburg, Russia, 2009, pp 24--35

Jones E, Abadi DJ, Madden S (2010) Low overhead concurrency control for partitioned main memory databases. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, 2010, pp 603--614

Jones EP, Abadi DJ, Madden S (2010) Low overhead concurrency control for partitioned main memory databases. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, 2010, June. pp 603--614

Kallman R, Kimura H, Natkins J, Pavlo A, Rasin A, Zdonik S, Jones EP, Madden S, Stonebraker M, Zhang Y, Hugg J, Abadi DJ (2008). H-store: a high-performance, distributed main memory transaction processing system. In Proceedings of the VLDB Endowment, 2008 Aug., 1(2) 1496--1499

Karger D, Lehman E, Leighton T, Panigrahy R, Levine M, Lewin D (1997) Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, 1997, pp 654--663

Kossmann D, Kraskan T, Loesing S (2010) An evaluation of alternative architectures for transaction processing in the cloud. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, 2010, June. pp 579--590

Kraska T, Hentschel M, Alonso G, Kossmann D (2009) Consistency rationing in the cloud: pay only when it matters. In Proc. VLDB Endow., 2009, Aug. 2(1) 253--264

Lahiri T, Srihari V, Chan W, MacNaughton N, Chandrasekaran S (2001) Cache Fusion: Extending Shared-Disk Clusters with Shared Caches. In Proceedings of the 27th international conference on Very Large Data Bases, 2001 pp 683--686

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

Lamport L (1978) Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. Communication ACM, 1978 July, 21(7). 558--565

Larson P, Blanas S, and etc (2011) High-performance concurrency control

mechanisms for main-memory databases. In Proceedings of the VLDB Endowment, 2011, Aug. 5(4) 298--309
Lewis P, Bernstein A, Kifer M (2002) Databases and transaction processing: an application-oriented approach. 2002, pp 764--773
Lin Y, Kemme B, Patiño-Martínez M, Jiménez-Peris R (2005) Middleware based data replication providing snapshot isolation. In Proceedings of the 2005 ACM SIGMOD international conference on Management of data, Baltimore, Maryland 2005, pp 419--430
Lloyd W, Freedman MJ, Kaminsky M, Andersen DG (2011) Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, 2011, pp 401--416
Manegold S, Kersten ML, Boncz P (2009) Database architecture evolution: mammals flourished long before dinosaurs became extinct. In Proceedings of the VLDB Endowment, 2009, Aug. 2(2) 1648--1653
MongoDB Website (2009) http://www.mongodb.org/. 2009
Olston C, Reed B, Srivastava U, Kumar R, Tomkins A (2008) Pig latin: a not-so-foreign language for data processing. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, Vancouver, Canada, 2008, pp 1099--1110
Oracle (2012) Technique overview of the Oracle Exadata Database Machine and Exadata Storage Server. Oracle White Paper. Oracle Co., Ltd 2012
Oracle RAC Website (2012) http://www.oracle.com/technetwork/products/clustering/overview/index.html. 2012
Özsu M (2011) Principles of distributed database systems. 2011, Chapter 11, 387--394
Poess M, Nambiar RO (2005) Large scale data warehouses on grid: Oracle database 10g and HP proliant servers. In Proceedings of the 31st international conference on Very Large Data Bases, Trondheim, Norway 2005 pp 1055--1066
Pritchett D (2008) BASE: An Acid Alternative. In ACM Queue, 2008 May, 6(3) 48--55
Ramamurthy R, DeWitt DJ, Su Q (2003) A case for fractured mirrors. In Proceedings of the VLDB Endowment, 2003, Aug. 12(2) 89--101
Rao J, Shekita EJ, Tata S (2011) Using Paxos to build a scalable, consistent, and highly available datastore. In Proc. VLDB Endowment, 2011, Jan. 4(4) 243--254
Roh H, Jeon M, Kim JS, Lee J (2011) Replicated abstract data types: Building blocks for collaborative applications. In J. Parallel Distrib. Comput., 2011, March, 71(3) 354--368
Ronstrom M, Thalmann L (2004) MySQL Cluster Architecture Overview. MySQL Technical White Paper, 2005, April.
Saito Y, Shapiro M (2005) Optimistic replication. In ACM Comput. Surv., 2005, Mar. 1(37) 42--81
Shapiro M, Preguiça N, Baquero C, Zawirski M (2011) Conflict-free replicated data types. In Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems, 2011, pp 386--400
Shute J, Vingralek R, Samwel B, and etc (2013) F1: A distributed SQL database that scales. In Proceedings of the VLDB Endowment, 2013, 6(11) 1068--1079
Shvachko K, Kuang H, Radia S, Chansler R (2010) The Hadoop Distributed File System. In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010, pp 1--10
Stonebraker M, Abadi DJ, Batkin A, Chen X, Cherniack M, Ferreira M, Lau E, Lin A, Madden S, O'Neil E, O'Neil P, Rasin A, Tran N, Zdonik S (2005) C-store: a column-oriented DBMS. In Proceedings of the 31st international conference on Very Large Data Bases, Trondheim, Norway 2005 pp 553--564
Stonebraker M, Madden S, Abadi DJ, Harizopoulos S, Hachem N, Helland P (2007) The end of an architectural era: (it's time for a complete rewrite). In Proceedings of the 33rd international conference on Very large data bases, Vienna, Austria, 2007, pp 1150--1160
Sybase Website (2010) http://sybase.com/. 2010
Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R (2009) Hive: a warehousing solution over a map-reduce framework. In Proceedings of the VLDB Endowment, 2009, Aug. 2(2) 1626--1629

Thusoo A, Shao Z, Anthony S, Borthakur D, Jain N, Sen Sarma J, Murthy R, Liu H (2010) Data warehousing and analytics infrastructure at facebook. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, Indianapolis, Indiana, USA, 2010, pp 1013--1020
Vertica Website (2011) http://www.vertica.com/. 2011
Vogels W (2009) Eventually Consistent. In ACM Queue, 2008, Oct. 6(6) 14--19
Voldemort Website (2011) http://project-voldemort.com/. 2011
VoltDB Website (2011) https://voltdb.com/. 2011
Weikum G, Vossen G (2001) Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery. 2001, pp 676--686
Welsh M, Culler D, Brewer E (2001) SEDA: an Architecture for Well-conditioned, Scalable Internet Services. In Proceedings of the 18th ACM symposium on Operating systems principles, Banff, Alberta, Canada. 2001, pp 230--243
Xu Y, Kostamaa P, Zhou X, Chen L (2008) Handling data skew in parallel joins in shared-nothing systems. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, 2008, pp 1043--1052
Yu H, Vahdat A (2002) Minimal replication cost for availability. In Proceedings of the twenty-first annual symposium on Principles of distributed computing, 2002, pp 98--107
Yu H, Vahdat A (2006) The costs and limits of availability for replicated services. In ACM Trans. Comput. Syst., 2006, Feb. 1(24) 70--113

Correspondence and offprint requests to: Lengdong Wu, Department of Computing Science, University of Alberta, Edmonton, AB, Canada. Email: lengdong@cs.ualberta.ca