Exploring the architecture of distributed database in both SQL and NoSQL communities.

Yiran Zhao, Xiaoxiao Wang
zhao97@illinois.edu, xwang104@illinois.edu
University of Illinois at Urbana – Champaign, Department of Computer Science

1. Abstract

The survey is aimed at understanding various design aspects of distributed database systems, and comparing the architecture and performance differences between relational database and non-relational database implementations.

First we investigate current techniques of database sharding in relational database systems, identify the challenges that come with increasingly large distributed storage systems, and find the intrinsic reasons for not being able to scale well.

Then, we investigate various types of non-relational database systems and study the design of popular NoSQL systems. The survey will analyze why non-relational database systems perform well in terms of availability and scaling, and highlights the most important features of current database systems. This survey will elaborate on the pros and cons of existing database systems and seek to find the best choice for different kinds of application scenarios.

Finally the hybrid database combining both MySQL and NoSQL features F1 is introduced at the end of this survey, and its salient design principle is discussed.

2. Motivation

For traditional relational database systems, it's easy to satisfy ACID (Atomicity, Consistency, Isolation, Durability) which guarantees that database transactions are processed reliably. With the advent of information overload, companies are now dealing with petabytes of heterogeneous data coming from everywhere and have to be stored in a distributed manner. Managing large collections of data distributed over different computers of a large computer network is very difficult to comply with strong ACID. Researches already found that providing extremely fast and parallel query processing and maintaining strong consistency of data is hard to achieve at the same time.

It's interesting to first find out techniques to deal with this problem on relational databases, and see why these approaches would be hard to further scale as the data exponentially increase. Then we investigate some of the industry's solutions of NoSQL database systems like MongoDB, Cassandra, Redis, Big Table, HadoopDB, etc., and seek to understand the advantage of such non-relational databases.

Additionally, our research topic as Ph.D. students is related to distributed databases, though not strictly the same. We plan to build web servers upon a cluster of energy-efficient small nodes, which would achieve similar performance requirements but consumes much less energy. Deploying database system on the small nodes would mean sharding data into finer pieces and designing management schemes to cope with limited per-node capacity. In other words, with larger cluster of wimpy nodes, database is also more distributed and achieving high availability and ACID is more difficult. This is why we want to delve into this topic.

3. Relational Database

Relational database combined with SQL language becomes a simple yet powerful declarative tool for set-oriented operations. SQL captures the essential patterns of data manipulation, including intersections, joins, filters, and aggregations or reductions. SQL's

declarative expressions are more readable and compact. Combining with SQL's GROUP BY operation, which is in reality a reduce function, SQL essentially provides the equivalence of operations such as those in the Map Reduce framework [8]. The major benefit of RDBMS is that it supports ACID, which is explained below.

3.1 Concept of ACID

ACID stands for Atomicity, Consistency, Isolation, Durability, where:

(1) Atomicity means that each transaction complies "all or nothing", i.e. if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged.

(2) Consistency means that if a transaction executed that violates the database's consistency rules, the entire transaction will be rolled back and the database will be restored to a state consistent with those rules. On the other hand, if a transaction successfully executes, it will take the database from one state that is consistent with the rules to another state that is also consistent with the rules.

(3) Isolation means that multiple transactions occurring at the same time do not impact each other's execution, as if they happen serially. But it does not ensure the order of transactions.

(4) Durability ensures that any transaction committed to the database will not be lost. Durability is ensured through the use of backups and transaction logs that can help restoring committed transactions in spite of subsequent software or hardware failures.

ACID is absolutely essential for most operational systems and online transaction processing systems, including retail, banking, and finance. ACID compliance may not be important to a search engine that may return different results to two users simultaneously, or to Amazon when returning sets of different reviews to two users. In these applications, speed and performance triumph the consistency of the results. However, in a banking application, two users of the same account need to see the same balance in their account. A utility company needs to display the same "payment due amount" to two or more users perusing an account. The idea of "eventual consistency" for such applications could lead to chaos in the business world.

3.2 Scalable RDBMS

Although relational database systems have intricate and bulky schemas, making it hard to distribute data across a large cluster of nodes, there are some techniques developed to achieve certain level of scalability without compromising ACID of RDBMS. Typically there are following approaches:

(1) Horizontal partitioning:

This approach partitions the database based on rows, i.e., a big table can be split horizontally according to the key of each row. To determine which node a row belongs to, there are several criteria. The simplest one would be key range partitioning. The target machine is determined just by a simple lookup. However, this is not efficient or load-balanced if new nodes join or some nodes leave. Another method is assigning destination according to the key's value. It's convenient if the keys' possible values can be enumerated, or listed. The third method is based on hashing. The output value of a carefully designed hash function determines the membership of the particular row. Consistent hashing is often used to evenly distribute data and workload among nodes.

(2) Vertical partitioning:

Vertical partitioning involves splitting a table up, separating the attributes or fields and store them as smaller tables across a cluster of nodes. If however, a group of attributes are

often selected together statistically, they should not be separated, or otherwise restoring the original tables too often due to particular queries will cost some performance penalty.

We will examine some of the examples in academia and industry that falls into these categories.

3.3 Horizontal Partitioning.

3.3.1 Examples and their features

(1) MySQL Cluster.

Traditionally, RDBMSs have not achieved the scalability required for today's data intensive applications. But as early as 10 years ago, MySQL Cluster [7] appeared the most scalable, although not achieving high performance per node compared to standard MySQL. MySQL Cluster horizontally partitions tables according the hash value of primary key. It is an ACID-compliant, multi-master architecture with no single point of failure [12].

MySQL Cluster works by replacing the InnoDB engine with a distributed layer called NDB. It is a technology providing shared-nothing clustering and auto-sharding, and designed to provide high availability and high throughput with low latency. MySQL Cluster achieves scalability by sharding data over multiple database servers, and supports crash recovery by replication. Bi-directional geographic replication is also supported. The replication in MySQL Cluster is synchronous, meaning that it guarantees that data is written to multiple nodes upon committing the data. However, between different clusters, like disaster replication across other data centers, the replication can be asynchronous. And in this case it uses optimistic control to detect and resolve conflict [12]. MySQL Cluster allows real-time response because it supports in-memory as well as disk-based data. It will only access disk storage to write redo records and checkpoints.

In MySQL Cluster, there are data nodes, which are storing sharded data and handling low level load-balancing, replication, failover and self-healing. Another type of node is management node, which is also the ndb_mgmd process. They are in charge of configuring and monitoring the cluster. The application node which runs mysqld process, connects to all of the data nodes in order to perform data storage and retrieval. This node type is optional, since there are other APIs like NoSQL API, NDB API, Memcached or REST, etc.

(2) VoltDB.

VoltDB is an in-memory database designed by several well-known database system researchers, It is an ACID-compliant RDBMS which uses a shared nothing architecture. Data and the processing associated with it are distributed among all the CPU cores. Data is durable to disk, which is ensured by continuous snapshots. Asynchronous command logging creates both snapshots and a log of all transactions between snapshots. And synchronous command logging writes transactions to the log after the transaction completes and before it is committed to the database. This ensures no transactions are committed that are not logged and no transactions are lost.

The reason that VoltDB is fast is that, first, database is stored in RAM on the servers, so that the system need never wait for the disk. Second, All SQL calls are made through stored procedures [7], with each stored procedure being one transaction. This means, if data is sharded to allow transactions to be executed on a single node, then no locks are required, and therefore no waits on locks. Transaction coordination is likewise avoided. Third, SQL execution is single-threaded for each shard, using a shared-nothing architecture, so there is no overhead for multi-thread latching.

This fast nature provides users with real-time analytics, real-time decision-makings with

millisecond latency. It achieves high-speed transactional ACID performance and the ability to process thousands to millions of incoming events per second. This is why VoltDB is now a successful industry product being used by big companies likes Mitsubishi, Ericsson, HP, etc.

(3) Clustrix.

Clustrix supports SQL with ACID-compliant transactions. Data is automatically sharded and replicated, and load balancing is transparent to the application programmer [7]. Clustrix is designed to be seamlessly compatible with MySQL, supporting existing MySQL applications and front-end connectors. This gives them a big advantage in gaining adoption of proprietary hardware.

Clustrix has built in fault-tolerance features for high availability, parallel backup and parallel replication among clusters for disaster recovery. The queries are also split up into fragments that are sent to nodes that own the required data. Another performance boost coming from the hardware side is the use of SSD, while from software side is the use of multi-version concurrency control and map-reduce work flow. However, Clustrix is based on proprietary software and hardware.

(4) ScaleDB.

Unlike other shared-nothing architecture, ScaleDB requires that the disks are shared across nodes, i.e., every server must have access to every disk. The entire dataset and schema is available to all the databases in the cluster and each database has read and write capabilities over the data. This allows it to operate concurrently over shared data to efficiently manage online transaction processing (OLTP) and data warehousing applications. This also allows it to have multi-table indexing, which can be beneficial when joining and ordering multiple tables on different servers. However, as a downside, this architecture has not scaled very well.

Nevertheless, it is ACID-compliant, and supports automatic sharding, allowing servers to be added freely, while also handling failure and automatically redistribute workload over the cluster. Shared disk also implies that there is no need to partition or shard the data when nodes join or leave, and high availability is automated as the system is built without a single point of failure.

3.3.2 Implementation using consistent hashing

Prior work solves the database hotspot problem using some kind of replication strategy that stores copies of hot pages throughout the Internet, and spreads the work of serving a hot page across several servers. So when a user makes a request for a data block, it is directed to an arbitrary server in the pool. If the data is stored there, it is returned to the user. Otherwise, the server forwards the request to all other servers via IP multicast. The disadvantage of this technique is that as the number of participating servers grows, even with the use of multicast, the number of messages the local network can become unmanageable, and potentially swamping servers. Consistent hashing makes this kind of communication not necessary all the time [13].

Borrowing the above idea and apply it to our own distributed RDBMS, we are thinking about doing it using just commodity computers, more precisely, small embedded system nodes available in our lab. The simplest way is to horizontally shard table rows across a cluster of nodes running MySQL. In order to consistently access a row given its key in an environment where nodes can be added or shutdown, we put the nodes on a virtual ring and assign key range according to their positions.

First given a key or id, we calculate the md5 value for this string key. Then the hash value is calculated according to the number of nodes in the ring. We also borrow the concept

from Chord [14] and Dynamo [11]. We virtually put the small nodes clockwise onto a ring-structured space, and having each node take charge of some key intervals according to their positions. For example, if the ring size is 256, then the hash value is the sum of every 8 bits in md5 hex number, and the row with that key will be stored on the node with position number immediately follows the hash value. Like Chord, the key range from a node's position to its successor's position will be stored on the successor node. In this way the rows are fairly evenly distributed across available nodes. However when nodes join or leave, the key range for each node is going to change and potentially cost performance problems. To alleviate that problem, we can create several virtual nodes for each physical node, and hash them to different places so that key range changes can be fairly equally balanced across existing nodes.

The number of copies can be specified upon creation of the ring. Typically it's 3 replications, stored on the target node, its successor and the successor of the successor. So in worst case it allows two servers to fail simultaneously. A master node would take care of the global view of the ring and handles queries to target node. If a target node is detected to be failed, the query would be passed to the successor.

This example, although simple, is easy to understand and implement. It also conveys important ideas found in papers like Chord [14] and Dynamo [11].

3.4 Vertical Partitioning.

3.4.1 Static Vertical Partitioning.

The problem of finding an optimal partitioning for a given workload and a given database schema is NP-complete problem [15]. So in general, vertical partitioning is grouped into two categories: overlapping and non-overlapping.

Non-overlapping means intersection of any two vertical partitions is empty (except the primary key of the table); otherwise it's overlapping partitioning. For non-overlapping partitioning, the storage required is less since there is no duplication. Furthermore, the cost for consistency control is less than for an overlapping partitioning. However, it would be hard to resolve queries accessing common attributes. On the other hand, queries are easier to be optimized in overlapping partitioning scheme, and since the cost to pay for disk is quite cheap, an acceptable redundancy may enhance the database performance dramatically. However, it would be harder to maintain consistency for overlapping partitions.

The A bottom-up approach for static vertical partitioning is to start with single-attribute partitions, and then gradually increase by merging the smaller partitions. The earliest solution involves affinity based clustering [16]. Affinity is used for the identification of the usage patterns, which later on, is applied to determine a representative workload. The affinity matrix is employed where each element is the cost determined by relevance or if they are frequently used together. The cost can be defined as the total number of read or write operations needed to compute a workload over a database schema. Another way is transaction based clustering [17]. Since the database transactions provide more semantic meanings than the attributes, it makes more sense to apply a transaction binary partitioning algorithm to decompose the relational schemas.

3.4.2 Dynamic Vertical Partitioning.

It refers to the type of partitioning that changes its schema according to the changing characteristics of database workload to automate performance tuning of relational database systems [15]. It needs a feed back control loop so that the workload is monitored and analyzed to optimize dynamic partitioning.

The algorithm of dynamic partitioning generally consists of a statistic collector and a

cluster analyzer. A statistic collector accumulates information about the queries run and the data returned, and it's responsible for triggering the cluster analyzer. Cluster analyzer determines the best possible clustering scheme given the statistics collected. If the new clustering is better than the current one then it triggers the reorganizer that physically re-organize the data on disk [18]. The statistic collector constructs an attribute affinity (AA) matrix. AA is a symmetric square matrix, which records the affinity among the attributes. The attribute affinity measures the strength of an imaginary bond between the two attributes, on the basis of the fact that attributes are used together by transactions. The second step is the clustering of attributes. The algorithm employs the bond energy algorithm (BEA) [21] and permutes rows and columns of a square matrix in order to obtain a semi-block diagonal form. The algorithm is typically applied to partition a set of interacting variables into different subsets, which interact minimally. The algorithms also clusters groups of attributes with high affinity in the same fragment, and keep attributes with low affinity in separate fragments.

To monitor the database and determine when to trigger the clustering, it is assumed that the workload of a database experiences sort of a cyclic pattern. For example, many companies are reviewing transactions at the early morning so that at daytime the workflow would be higher. Also the type of query can appear regularly at specific time according to the type of the workload. Thus by analyzing historical query data we can deploy optimal partitioning strategy at low workload time and embrace the anticipated heavy workload time, so as to optimally balance database workload and prevent hotspots appearing.

But patterns are not necessarily on a daily time scale, as it can be on any smaller scales, which requires a proper way to find similarities between workloads. So given the contents of trace files, the similarity of workload between two periods are calculated. Similarity can be derived from comparing the list or set of sessions between two workloads, where session can be defined as a sequence of executions in a certain time period [15]. So in a trace file representing a continuous time workload, signatures of each time unit of workload is calculated and compared with the current workload signature. If the current workload is a cyclic point and also a low workload point, re-configuration can be employed at this time according to anticipate trend.

Compared to static vertical partitioning, dynamic may not always find the optimal partition scheme due to the difficulty to gain sufficient information about future workload characteristics. However, static long-term partition scheme is unlikely because the applications are constantly changing.

3.5 Summary

RDBMS has a complete pre-defined schema, a SQL interface, and ACID properties. The benefits of relational database also cause it to be hard to scale. It appears likely that some relational DBMSs like VoltDB or Clustrix will provide scalability comparable with NoSQL data stores, with two provisos:

(1) Use small-scope operations. Operations that span many nodes, e.g. joins over many tables, will not scale well with sharding.

(2) Use small-scope transactions. Transactions that span many nodes are going to be very inefficient, with the communication and two-phase commit overhead.

However, even with the two conditions, relational distributed database still lags behind new NoSQL systems in terms of availability, scalability and fault-tolerance. But we will see in the next section that actually NoSQL systems avoid these two problems by making it difficult or impossible to perform larger-scope operations and transactions. In other words, a relational DBMS makes expensive multi-node multi-table operations easy, while NoSQL systems make them impossible or obviously expensive for programmers.

In this perspective, a scalable RDBMS should perform worse in some cases because it

does not need to preclude larger-scope operations and transactions. Instead, they simply penalize users for these operations if they use them.

4. NoSQL

Today's landscape of data management applications is much more diverse than when the relational model was born. Examples of this diversity are: semi-structured data, unstructured data, continuous data, sensor data, streaming data, uncertain data, graph data, etc. In additional to data diversity, there is information overload, where processing and retrieving huge amount of data in low latency seems more important than ever. In large scale and highly concurrent applications, using relational database is becoming inadequate. Thus common intuition is to sacrifice consistency for better availability and scalability. This gives rise to NoSQL as a new version of database system tuned for today's big data applications.

4.1 Advantages of NoSQL

Compared to traditional RDBMS, NoSQL has the following features:

(1) High concurrency of reading and writing with low latency.

NoSQL is designed to meet the need of highly concurrent reading and writing with low latency at the same time, in order to enhance customer satisfaction. Large applications, such as SNS and search engines, need database to meet the efficient data storage (PB level) and can respond to millions of requests fast [19].

(2) High scalability and high availability.

With the increasing number of concurrent requests, the database needs to be deployed on large clusters of servers and to be able to support easy expansion and upgrades, and ensure rapid uninterrupted service.

(3) Low management and operational costs.

With the dramatic increase in data volume, database costs, including hardware costs, software costs and operating costs, have dramatically increased. Therefore, designing database on commodity and cheap machines is important.

4.2 Types of NoSQL.

4.2.1 Key-value database

Examples are Riak, Dynamo, Redis, Dbm, Tokyo Cabinet and Kyoto Cabinet, Flare, etc.

Data is represented in key-value pairs, and each key only appear once in the collection. The key is typically a string while the value can be String, JSON, BLOB (basic large object) etc. While key-value based store involves least complexity and enjoys the most availability and scalability, it does not support any traditional database capabilities (atomicity, or consistency). Such capabilities must be provided by the application itself.

4.2.2 Document-oriented database

Examples are MongoDB, CouchDB, Clusterpoint, Couchbase, DocumentDB, HyperDex, etc.

Document oriented databases differ from key-value based store in that they provide some structure and encoding of the managed data. XML, JSON, BSON (binary encoding of JSON objects) are common standard encodings. Because of the document, schema-less style, changing or adding fields to the documents is easy. It also tolerates incomplete data, but has no standard query syntax.

4.2.3 Column-oriented database

Examples are HBase, HadoopDB, Cassandra, Hypertable, Bigtable, PNUTS, etc.

Data is stored in columns rather than in rows, and columns that are often used together could be grouped into column families. Columns can be defined at run time, unlike the rigid schema that need to be defined beforehand as in SQL. The benefit of storing data in columns is fast search, access and data aggregation, because cells in continuous columns are stored together. However, weakness of this category is that it usually has very low-level API.

4.2.4 Graph-based database

Examples are IBM DB2, InfiniteGraph, InfoGrid, Neo4j, OrientDB, etc.

In this category, data is stored and represented as nodes, and their relationship as edges. Graph databases excel at managing highly connected data and complex queries. Given a set of starting points, graph databases can explore the larger neighborhood around the initial starting points, collecting and aggregating information from millions of nodes and relationships. Graph databases are often faster for associative data sets.

4.3 Examples of NoSQL and their features.

4.3.1 Dynamo [11]

According to Brewer's theorem, consistency, availability and partition tolerance can not be achieved at the same time. Amazon's Dynamo chooses A+P and sacrifices consistency, because 1) for e-commerce, even small outage is costly; 2) an "always-on" experience for users is important. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services [11].

1) Service requirements:

Dynamo works best when services only need primary-key access to a data store. It assumes that no operations span multiple data items and there is no need for relational schema. Many services, like those providing bestseller lists, shopping carts, customer preferences, session management, sales rank, and product catalogs, have this feature. The common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications. Dynamo does not provide any isolation guarantees and permits only single key updates.

2) System structure:

Dynamo uses consistent hashing described in Section 3.3.2, where each node is assigned random positions on the ring structure and takes charge of the keys whose value falls into the range of each node's position and the previous node's. To more evenly distribute loads, each node acts as several virtual nodes, according to its capacity and heterogeneous physical infrastructure. Each data is replicated N times, stored in the successors in the ring. Quorum based techniques and decentralized protocol ensure the consistency of replicas. And failure detection is realized using gossip based distributed membership protocol. These allow for arbitrary join and leaving of servers without any manual partitioning or redistribution.

The Dynamo ring structure partitions the ring space into equal sized "buckets", and each node takes equal amount of partitions. This decouples data partitioning from data placement, with benefits such as fewer overheads when nodes leave and join, easier to archive data chunks, and faster bootstrapping and recovering. All these techniques make Dynamo a highly available and scalable data store.

3) Design highlights:

In the design, service level agreements are expressed and measured at the 99.9th percentile of the delay distribution, since the goal is to build a system where all customers have a good experience, rather than just the majority. One of the main design considerations

for Dynamo is to give services control over their system properties, such as durability and consistency, and to let services make their own tradeoffs between functionality, performance and cost- effectiveness. Dynamo targets the design space of an “always writeable” data store (highly available for writes), since rejecting customer updates could result in a poor customer experience. So with respect to when to resolve conflicts, Dynamo chooses reads. And as for who resolves conflicts, it’s application, since the application is aware of the data schema and can decide on the conflict resolution method that is best suited for its client’s experience.

The read/write requests are routed either using a generic load balancer that select nodes based on the load, or a client library that routes the request to appropriate nodes. Typically only one of the nodes in the preference list will eventually handle the request. To maintain consistency among replicas, a quorum based system where the minimum number of nodes must participate for read operation is R and for write is W (R+W>Nodes in preference list). To maintain SLA, the coordinator for a write is chosen to be the node that replied fastest to the previous read operation, which is stored in the context information of the request. The main advantage of Dynamo is that its client applications can tune the values of N, R and W to achieve their desired levels of performance, availability and durability. For instance, the value of N determines the durability of each object. A typical value of N used by Dynamo’s users is 3, and R and W are 2.

4) Usage:

Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. This is why Amazon’s global e-commerce platform uses this database to satisfy various kinds of applicatioins.

4.3.2 Bigtable [6]

Bigtable is a distributed storage designed for managing large-scale structured data, while meeting different demands of various services. Some services such as web indexing require small data size, while others like Google Earth need to store big satellite images. Latency demands are also different, with some processing in real-time and others doing batch processing in background. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

1) Data model:

Bigtable does not support a full relational data model. Instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk. A table in Bigtable is a sparse, distributed, persistent multidimensional sorted map. Data is organized as the mapping: (row, column, timestamp) -> string.

Row: In Bigtable, row is the unit of transactional consistency and it does not currently support transactions across rows. Rows with consecutive keys are grouped into tablets, which form the unit of distribution and load balancing. As a result, reads of short row ranges are efficient and typically require communication with only a small number of machines.

Column: Column keys are grouped into sets called column families, which form the unit of access control. All data stored in a column family is usually of the same type, which makes it easier to compress. Although a table may have an unbounded number of columns, the number of distinct column families in a table is small (at most in hundreds), and column families rarely change during operation. This keeps widely shared metadata from being too large.

Timestamp: Different cells in a table can contain multiple versions of the same data, where the versions are indexed by timestamp. Timestamps can be assigned implicitly by Bigtable, in which case they represent "real time" in microseconds. Or they can be assigned explicitly by client applications. Timestamps are also used in garbage-collection, where client can specify that only last n versions be kept, or only versions later than a time be kept.

2) Storage design:

Bigtable uses Google File System [23] to store log and data files. GFS is a distributed file system that maintains multiple replicas of each file for greater reliability and availability. The Google *SSTable* immutable file format is used internally to store Bigtable data files. Internally, each SSTable contains a sequence of blocks. A block index is used to locate blocks, which is loaded into memory when the SSTable is opened. A lookup can be performed by first finding the appropriate block using binary search in the in-memory index, followed by reading the appropriate block from disk. Optionally, an SSTable can be completely mapped into memory, which allows us to perform lookups and scans without touching disk.

Initially, each table consists of just one tablet. As a table grows, it is automatically split into multiple tablets, each approximately 1 GB in size by default. A tablet server commits a split by recording information for the new tablet in the METADATA table. The metadata about tablets are stored in the hierarchical structure like B+ tree, with the root table storing all the locations of metadata tables. The client library traverses the location hierarchy to locate tablets, and caches the locations that it finds.

Bigtable preserves the fast read access which SSTable gives us, but we also want to support fast random writes. Random writes are fast when the SSTable is in memory (MemTable), and the table is immutable, meaning an on-disk SSTable is also fast to read from. So, SSTable indexes are always loaded into memory, and all writes go directly to the MemTable index. Reads first check the MemTable and then the SSTable indexes. Periodically, the MemTable is flushed to disk as an SSTable and on-disk SSTables are collapsed together.

3) System structure:

The Bigtable implementation has three major components: a library that is linked into every client, one master server, and many tablet servers. The master is responsible for assigning tablets to tablet servers, detecting the addition and expiration of tablet servers, balancing tablet-server load, and garbage collecting files in GFS. Because Bigtable clients also do not rely on the master for tablet location information, most clients never communicate with the master. As a result, the master is lightly loaded in practice.

Bigtable relies on a highly available and persistent distributed lock service called Chubby. Chubby uses the Paxos algorithm to keep its replicas consistent in the face of failure. Bigtable uses Chubby for a variety of tasks: 1) to ensure that there is at most one active master at any time; 2) to store the bootstrap location of Bigtable data; 3) to discover tablet servers and finalize tablet server deaths; 4) and to store Bigtable schemas. When a tablet server starts, it creates and acquires an exclusive lock on a uniquely named file in a specific Chubby directory. When it leaves, it releases its lock so that the master will reassign its tablets more quickly. Chubby is an effective communication substrate for Bigtable schemas because it provides atomic whole-file writes and consistent caching of small files.

Fast read and write, plus high scalability, make Bigtable suitable for many products like Google Earth, Personalized Search and Google Analytics, and for more than sixty projects as of August 2006. However, its usage interface may cause some problems with people accustomed with relational database language. Nevertheless, the facts suggest that Bigtable works well in practice.

4.3.3 Cassandra [9]

Cassandra is like a combination and evolution of Dynamo and Bigtable. It's a distributed storage system for managing very large amounts of structured data, while providing highly available service with no single point of failure.

1) Data model (similar to Bigtable):

Cassandra does not support a full relational data model. Instead, it provides clients with a simple data model that supports dynamic control over data layout and format. Similar to Bigtable, every operation under a single row key is atomic per replica no matter how many columns are being read or written into. And columns are grouped together into sets called column families.

Particularly to Cassandra is that it exposes two kinds of columns families, Simple and Super column families. Super column families can be visualized as a column family within a column family. Furthermore, applications can specify the sort order of columns within a Super Column or Simple Column family. The system allows columns to be sorted either by time or by name within a Super Column or Simple Column family.

2) Data partitioning (similar to Dynamo):

Cassandra partitions data using a hash function that is order preserving. The basic consistent hashing algorithm presents some challenges. First, the random position assignment of each node on the ring leads to non-uniform data and load distribution. Second, the basic algorithm is oblivious to the heterogeneity in the performance of nodes. Typically there exist two ways to address this issue: One is for nodes to get assigned to multiple positions in the circle, like in Dynamo, and the second is to analyze load information on the ring and have lightly loaded nodes move on the ring to alleviate heavily loaded nodes. Cassandra opts for the latter as it makes the design and implementation very tractable and helps to make very deterministic choices about load balancing.

Similar to Dynamo, replication is specified by a factor N and replicas are stored in the successors of coordinator (primary node for each key). But Cassandra provides various replication policies such as "Rack Unaware" (as in Dynamo), "Rack Aware" and "Datacenter Aware". For the latter two strategies the algorithm is slightly more involved. Cassandra system elects a leader amongst its nodes using a system called Zookeeper [24]. All nodes on joining the cluster contact the leader who tells them for what ranges they are replicas for and leader makes a concerted effort to maintain the invariant that no node is responsible for more than N-1 ranges in the ring. This means every node is aware of every other node in the system and hence the range they are responsible for. And also Cassandra uses preference list to store the nodes that stores the key.

3) Membership management:

Membership in Cassandra is based on Scuttlebutt [25], a very efficient anti-entropy Gossip based mechanism. The salient feature of Scuttlebutt is that it has very efficient CPU utilization and very efficient utilization of the gossip channel. Failure detection is used to avoid attempts to communicate with unreachable nodes during various operations. Cassandra uses a modified version of the Accrual Failure Detector [26]. The idea is that the failure detection module doesn't emit a Boolean value stating a node is up or down. Instead the failure detection module emits a value, which represents a suspicion level for each of monitored nodes.

When a node joins the ring, it randomly chooses a token as its position and gossips this information to other nodes. Initially there are contact nodes in the ring that give new nodes configuration files, so they are called seeds in the cluster. Since node failure can be transient, a node's outage rarely signifies a permanent departure and therefore should not result in re-balancing of the partition assignment or repair of the unreachable replicas. Similarly, manual error could result in the unintentional startup of new Cassandra nodes. For these

reasons, it was deemed appropriate to use an explicit mechanism to add and remove nodes from a Cassandra instance.

4) Read/write operations:

Write operation involves a write into a commit log for durability and recoverability and an update into an in-memory data structure. The write into the in-memory data structure is performed only after a successful write into the commit log. When the in-memory data structure crosses a certain threshold, calculated based on data size and number of objects, it dumps itself to disk. This process is very similar to the compaction process that happens in the Bigtable system. A typical read operation first queries the in-memory data structure before looking into the files on disk. In order to prevent lookups into files that do not contain the key, a bloom filter, summarizing the keys in the file, is also stored in each data file and also kept in memory. In order to prevent scanning of every column on disk, column indices are used to jump to the right chunk on disk for column retrieval. Cassandra morphs all writes to disk into sequential writes thus maximizing disk write throughput. Since the files dumped to disk are never mutated no locks need to be taken while reading them.

Cassandra is the daughter of Bigtable and Dynamo, borrowing partitioning concepts from Dynamo and data model from Bigtable. Cassandra has implemented its own CQL (Cassandra Query Language), the syntax of which is obviously modeled after SQL. It's relatively easy to install, setup and use and offers tunable consistency in addition to high availability and scalability. That's why Cassandra is widely used by companies like Facebook, Netflix, Instagram, DataStax, Twitter, etc.

4.3.4 Spanner [20]

Spanner is a newer version of Google's distributed system, and it's globally distributed, supporting synchronous replication and externally consistent distributed transactions. It is a database that shards data across many sets of Paxos state machines in data centers all over the world. Spanner is designed to scale up to millions of machines across hundreds of datacenters and trillions of database rows. Spanner's main focus is managing cross-datacenter replicated data, but it also improves consistency by adding multi-version into database. Data is stored in schematized semi-relational tables, and is versioned and automatically time stamped with its commit time. Spanner supports general-purpose transactions, and provides an SQL-based query language.

1) Highlighting features:

Spanner provides applications with huge flexibility in choosing reading and writing latency requirement, by allowing applications to choose which data centers to store the replicas and how many replicas are there. Surprisingly, Spanner provides externally consistent reads and writes, and globally consistent reads across the database at a timestamp. This enables consistent backups, consistent MapReduce executions, and atomic schema updates, all at global scale, even in the presence of ongoing transactions. This external consistency is enabled by the design and implementation of the TrueTime API, which directly exposes uncertainty of commit timestamps. And the logic is that if the uncertainty is large, the Spanner will slow down to wait out that uncertainty to make sure that the boundaries between the commit timestamps are clear. Spanner keeps uncertainty small (generally less than 10ms) by using multiple GPS and atomic clocks.

2) System structure:

Spanner platform is divided into a few major universes, such as playground universe, development universe and production-only universe. Each universe consists of many zones, which are similar to clusters of Bigtable servers. Each zone has one zone-master and hundreds of thousands of spanservers, and some location proxies. The universe master is a console for interactive debugging, and placement driver for a zone handles automated

movement of data across the zones. Each spanserver is responsible for hundreds of tablets, which is similar to Bigtable's tablet abstraction. And each spanserver implements a Paxos state machine on top of each tablet, with the leader under time-based leases for 10 seconds. The reason for this long living leader is that proper managing of the locking of tables needs it during the two-phase locking.

3) Enabling transactional semantics:

Unlike Bigtable, Spanner assigns timestamps to data, which is an important way in which Spanner is more like a multi-version database than a key-value store. Writes must initiate the Paxos protocol at the leader, while reads access key-value mapping state directly from the underlying tablet at any replica that is sufficiently up-to-date. The set of replicas is collectively a Paxos group. Each leader (spanserver) implements a lock table that contains the state for the two-phase locking on the primary replica. The leader of the replica also implements a transaction manager that is used to manage distributed transactions. If a transaction involves only one Paxos group, it can bypass the transaction manager, since the lock table and Paxos together provide transaction property. If a transaction involves more than one Paxos group, those groups' leaders coordinate to perform two-phase commit. The state of each transaction manager is stored and replicated in its Paxos group.

Spanner uses supporting directories to allow applications to control the locality of their data, where a directory is a set of continuous keys that share a common prefix. Directory is the unit used in data moving between zones, and replication configuration within a directory is the same. Moving directories helps load balancing, better locality, and put directories that are frequently accessed together into the same group.

Bigtable lacks cross-row transactions, which often lead to complaints from users. That's why many Google applications like Gmail, Android Market, AppEngine uses Megastore to manage consistency. But Spanner is built to mitigate this problem. It gives the application the responsibility to tradeoff between availability and transaction properties. Running two-phase commit over Paxos increases the availability despite the locking.

4) Most important design: TrueTime API

This is a very powerful API in Spanner, where the time is represented by a bounded time interval with explicit uncertainty. TT.now() returns a time interval that guarantees the true timestamp lies within this bounded interval, i.e., TT.now().earliest<=Tnow<=TT.now().latest, where Tnow is the time of invocation event. The underlying time references used by TrueTime are GPS and atomic clocks. TrueTime uses two forms of time reference because they have different failure modes, that is, atomic clocks can fail in ways uncorrelated to GPS and each other.

Each data center has time master machines equipped with either physically separated GPS receivers, or atomic clocks. Time masters compare their time reference with each other, and also with its own local clock. Daemons poll a variety of masters and apply a variant of Marzullo's algorithm to detect liars and synchronize with non-liars. Usually the uncertainty is within 7ms, however, the occasional master unavailability can cause datacenter-wide increase in time uncertainty.

5) Usage:

Google's advertising backend F1 uses Spanner to store data, although previously it uses sharded MySQL database. Resharding the revenue-critical database as it grew in the number of customers was extremely costly. This is why the team had to limit growth on the MySQL database by storing some data in external Bigtable, which compromised transactional behavior and the ability to query across all data. Finally the F1 team chose to use Spanner for several reasons. First, Spanner removes the need to manually re-shard. Second, Spanner provides synchronous replication and automatic failover. Finally, F1 needs strong transactional semantics, which make other NoSQL databases impractical.

4.3.5 MongoDB [10]

MongoDB is an open source non-relational document-based database written in C++. Although MongoDB is NoSQL, it implements many features of relational databases, such as sorting, secondary indexing and range queries.

1) Data model:

MongoDB does not organize data in tables with columns and rows. Instead, data is stored in "documents", each of which is an associative array of scalar values, lists, or nested associative arrays. MongoDB documents are serialized naturally as JavaScript Object Notation (JSON) objects, and are in fact stored internally using a binary encoding of JSON called BSON [6]. The identification of records is made by defined type, not just id. For example, it can be the combination of id and timestamp in order to keep documents unique. It is important to notice that 32bit MongoDB has a major limitation. Only 2GB of data can be stored per node. The reason of that is memory usage made by MongoDB. In order to increase performance data files are mapped in memory. By default data is sent to disc every 60 seconds but that time can be personalized.

In order to increase performance while working with documents, MongoDB uses indexing similar to relational databases. Each document is identified by the "_id" field, and additional indexes can be created by database administrator. All indexes use the B-tree structure. Each query uses only one index chosen by query optimizer, giving preference to the more efficient index. Eventually query optimizer reevaluates used indexing by executing alternative plans and comparing execution cost.

2) Scalability with auto-sharding:

To scale its performance on a cluster of servers, MongoDB uses a technique called auto-sharding, which is the process of splitting the data evenly across the cluster to parallelize access. Auto-sharding provides automatic balancing for changes in load and data distribution, easy addition of new machines without down time, automatic fail recover and eliminating single point of failure. Parallel access is achieved by dividing the MongoDB server into a set of front-end routing servers (mongos), which route operations to a set of back-end data servers (mongod).

MongoDB uses a form of range partitioning scheme to distribute records instead of hash-based partitioning. To partition a collection, MongoDB specifies a sharding key pattern which has one or more fields to define the key upon which we distribute data. Chunks can be described as a collection with min key and max key. When a chunk grows to a maximum size, usually 200MB, it splits into two new chunks. When the number of chunks between two servers grows to a certain threshold, the balancer will redistribute the chunks until the number of chunks is even across all servers.

3) Map/Reduce, flexible aggregation and data processing:

MongoDB queries examine one record at a time, which means that queries across multiple records must be implemented on the client or use MongoDB's built-in MapReduce (MR). Though MongoDB's MR can be executed in parallel at each shard, there are two major drawbacks: (1) the language for MR scripts is JavaScript, which is slow and has poor analytics libraries, and (2) the JavaScript implementation used by MongoDB is not thread-safe, so only one MapReduce program can run at a time.

4) Limitations:

Since MongoDB uses up more memory to store the key names within each document and map data files in memory, when the data set grows too large, its performance degrades fast compared to other NoSQL like Cassandra.

MongoDB uses a readers-writer lock that allows concurrent reads access to a database,

but gives exclusive access to a single write operation. When a write lock exists, a single write operation holds the lock exclusively, and no other read or write operations may share the lock. This may cause some performance issues compared to highly writable database like Dynamo.

To compare with RDBMS, there is no join operation in MongoDB. Also, MongoDB doesn't automatically treat operations as transactions. In order to ensure data integrity upon create/update you have to manually choose to create a transaction, manually verify it, and then manually commit or rollback. This creates some trouble for users who want some transaction semantics and join operations.

4.4 Summary of NoSQL data-stores.

Bigtable excels at dealing with datasets that start at several Terabytes. It is an ideal database to analyze the data in massive aggregate scale while the requests are also massive. Thus it is used to run MapReduce and various big data analysis toolkits on production sites intensely after it's born. But query capabilities are limited and access control mechanisms are not sufficient. Even though many projects happily use Bigtable, developers also consistently complains that Bigtable can be difficult to use for applications that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication.

Compared to Bigtable, Dynamo targets applications that require only key/value access with primary focus on high availability where updates are not rejected even in the wake of network partitions or server failures. It is built for its largest e-commerce website and aims at providing customers with smooth shopping experiences, thus highly writable and high availability are its features. But a few services are also exploring Dynamo's quorum characteristics and using it as a high performance read engine.

Cassandra is great for storing time series data, since each row can contain almost unlimited columns and the values can be sorted by timestamp. By providing a tunable consistency model, there is a greater flexibility for the developer and architect. A new feature called Lightweight Transactions (LWT) was released in Cassandra 2.0. This brings tunable ACID capabilities to Cassandra. Tunable consistency is about choice, and developers and architects can have those choices. Cassandra offers solution for problem where requirement is to have very heavy write operations and you want to have quite responsive reporting system on top of the stored data.

Spanner utilizes new technology: TrueTime enabled by atomic clocks and GPS receivers to provide global consistency and transaction semantics while achieving all other NoSQL high availability and scalability. But Spanner is primarily used by Google F1 SQL database management system (DBMS) and is not for public.

MongoDB is relatively popular among NoSQL database family. It is fast, flexible and publicly available, and often used in web applications such as micro blog websites. Although it has internal auto-sharding mechanisms, MongoDB still lag behind when dealing with extremely large datasets, where Cassandra excels more. Nevertheless, MongoDB is still under development and has great potential and a bright future.

5. Hybrid database

5.1 F1 [2]

F1 is a hybrid database that combines high availability, scalability of NoSQL systems like Bigtable, and the consistency and usability of traditional SQL databases. F1 is built on Spanner, which provides synchronous cross-datacenter replication and strong consistency. Previously Google's AdWords system uses sharded MySQL database but it failed to meet their requirements for scalability and reliability. Using the features of Spanner and Bigtable, F1 is able to combine the availability of the latter and the consistency of Spanner brought by TrueTime API. This allows F1 to never have down time and provide necessary ACID

transactions, and most importantly to use full SQL query support.

1) Features:

- Distributed SQL queries, including joining data from external data sources. Allowing join operation is a dream for developers, and is necessary in some types of applications. This feature makes MongoDB and Cassandra less attractive.
- Transactional consistent secondary indexes. Globally consistent timestamp enables transaction semantics across data centers.
- Asynchronous schema changes including database re-organizations.
- Optimistic transactions.
- Automatic change history recording and publishing.

2) Mitigating higher latency:

The above features inevitably result in higher read and write latency. F1 uses a hierarchical schema model with structured data types and through smart application design.

First, users interact with F1 through F1 clients, and the F1 client and load balancer prefer to connect to an F1 server in a nearby datacenter whenever possible, only otherwise if the load of local servers is too high or in case of failures.

Second, F1 servers are typically co-located in the same set of datacenters as the Spanner servers storing the data. This co-location ensures that F1 servers generally have fast access to the underlying data. The Spanner servers in each datacenter in turn retrieve their data from the Colossus File System (CFS) in the same datacenter, and will not communicate with remote CFS systems.

Third, F1 servers are mostly stateless, allowing a client to communicate with a different F1 server for each request. This benefits load balancing as well as making F1 server joining or leaving easier since no data movement will ensue. One exception is that when a user is using pessimistic transactions and is holding locks, it is then bound to one F1 server for the duration of that transaction.

Forth, SQL query execution is made distributed instead of centralized when the query planner estimates that increased parallelism will reduce query-processing latency. F1 also supports large-scale data processing through MapReduce, and MapReduce servers are allowed to communicate directly with Spanner servers to extract data in bulk to reduce time. While other clients perform reads and writes exclusively through F1 servers.

Fifth, when executing queries involving data from remote machines, F1 employs batching or pipelining data access to mitigate network latency. F1's network based storage is typically distributed over many disks, because Spanner partitions its data across many physical servers. This results in near-linear speedup using parallel access and avoids the contention for the same resources.

3) Data model:

Logically tables in F1 schema are organized into a hierarchy. Physically, F1 stores each child table clustered with and interleaved within the rows from its parent table. To put child tables interleaved with parent table, the child table must have a foreign key to its parent table as a prefix of its primary key. The hierarchically clustered physical schema has several advantages over a flat relational schema. Since child tables are often retrieved with parent tables together during a query, placing them in the same physical cluster can facilitate common-case join processing. Because the tables are both stored in primary key order, rows from the two tables can be joined using a simple ordered merge, and since the data is clustered into a single directory, we can read it all in a single Spanner request.

Hierarchical clustering is especially useful for updates, since it reduces the number of Spanner groups involved in a transaction. Because each root row and all of its descendant rows are stored in a single Spanner directory, transactions restricted to a single root will usually avoid 2PC and the associated latency penalty, so most applications try to use

single-root transactions as much as possible. In AdWords, most transactions are typically updating data for a single advertiser at a time, so we made the advertiser a root table (Customer) and clustered related tables under it.

4) Schema changes:

F1 as a business platform is shared by thousands of users and making schema changes non-blocking is very important. Making synchronous changes across all servers would be disruptive to this goal. So F1 schema changes are applied asynchronously, on different F1 servers at different times. To prevent database corruption, it is enforced across all F1 servers that at most two different schemas are active at any time. Leases are granted on the schema to ensure that no server uses a schema after lease expiry. F1 also subdividing each schema change into multiple phases where consecutive pairs of phases are mutually compatible and cannot cause anomalies.

5) Transactions:

A transaction consists of multiple reads, followed by a write that commits the transaction. F1 implements three types of transactions: read-only snapshot transactions, pessimistic transactions and optimistic transactions. For the latter two, there is a hidden stored last modification timestamp for each row. When the client commits, the F1 server will re-read the last modification timestamps for all read rows, and if any of the re-read timestamp differs from the one passed to the client, the transaction involving the rows is aborted. Otherwise the F1 sends the writes to the Spanner along with the commit timestamp being stored in the hidden column to commit the transaction.

5.2 Summary

Data stores combining MySQL ACID with NoSQL scalability and availability are very rare, with F1 being the most prominent one in production use. The transactional capability is inherited from the globally consistent timestamp in Spanner combined with a hierarchical schema design, and the scalability is inherited from the Bigtable and parallel data streaming. While executing some large OLAP queries in F1 experiences similar delay to MySQL database, its availability is beyond any distributed MySQL databases. Most importantly F1 enables SQL queries that make it easy for developers to use, which is why Google's core AdWords business is now running completely on F1.

6. Related work

For sharded MySQL databases, [3] introduces some SQL sharding scheme such as how to group and home data (ranges vs hash vs lookup table). Sharding need to consider application, service, and operational requirements. We will focus on issues like attribute partitioning using heuristic based approach or dynamic partitioning approach.

[8] discusses the evolution of data management landscape, lists the reasons for the decline of the relational model and of SQL and the rise of the non-relational technology. It also discusses the possible consequences of sacrificing the ACID properties in favor of system performance and data availability.

[7] gives us a comprehensive comparison of current implementations of SQL and NoSQL database systems, and discusses how they're designed to scale simple OLTP-style application loads over many servers. [5] conducts performance and scaling experiments on RDBMS vs. NoSQL and finds out that MongoDB can perform much better for complicated queries at the cost of data duplication which in turn results to a larger database; MySQL performs best at deletion whereas MongoDB excels at inserting documents; using sharding to split up the database in MongoDB did not provide a performance advantage which may be related to the routing done by the MongoDB system. [1] explores performance and scalability aspects of NoSQL vs. SQL on two classes of workloads: interactive data-serving

environments and decision support systems (DSS). They believe SQL systems will need to expand their functionality (for example, supporting automatic sharding and a more flexible data model such as JSON) to continue to be competitive.

NoSQL databases such as Bigtable [6], Cassandra [9], HBase [4], MongoDB [10] and Dynamo [11] each has their features and was designed in favor of the company's business model. Lastly this survey introduces F1 [2], the middle ground between the two types of databases and explores its architecture and mechanism that enable both consistency and availability.

Reference:

[1] Floratou, Avrilia, Nikhil Teletia, David J. DeWitt, Jignesh M. Patel, and Donghui Zhang. "Can the elephants handle the NoSQL onslaught?" Proceedings of the VLDB Endowment 5, no. 12 (2012): 1712-1723.

[2] Shute, Jeff, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea et al. "F1: A distributed SQL database that scales." Proceedings of the VLDB Endowment 6, no. 11 (2013): 1068-1079.

[3] Evan Elias, "Tumblr.com: Massively Sharded MySQL", Velocity Europe 2011.

[4] Apache Foundation. Apache HBase. http://hbase.apache.org/.

[5] Hadjigeorgiou, Christoforos. "RDBMS vs NoSQL: Performance and Scaling Comparison." The University of Edinburgh (2013).

[6] Chang, Fay, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A distributed storage system for structured data." ACM Transactions on Computer Systems (TOCS) 26, no. 2 (2008): 4.

[7] Cattell, Rick. "Scalable SQL and NoSQL data stores." ACM SIGMOD Record 39, no. 4 (2011): 12-27.

[8] Atzeni, Paolo, Christian S. Jensen, Giorgio Orsi, Sudha Ram, Letizia Tanca, and Riccardo Torlone. "The relational model is dead, SQL is dead, and I don't feel so good myself." ACM SIGMOD Record 42, no. 2 (2013): 64-68.

[9] Lakshman, Avinash, and Prashant Malik. "Cassandra: a decentralized structured storage system." ACM SIGOPS Operating Systems Review 44, no. 2 (2010): 35-40.

[10] Liu, Yimeng, Yizhi Wang, and Yi Jin. "Research on the improvement of MongoDB Auto-Sharding in cloud environment." In Computer Science & Education (ICCSE), 2012 7th International Conference on, pp. 851-854. IEEE, 2012.

[11] DeCandia, Giuseppe, et al. "Dynamo: amazon's highly available key-value store." ACM SIGOPS Operating Systems Review. Vol. 41. No. 6. ACM, 2007.

[12] https://en.wikipedia.org/wiki/MySQL_Cluster

[13] Karger, David, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. "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, pp. 654-663. ACM, 1997.

[14] Stoica, Ion, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. "Chord: A scalable peer-to-peer lookup service for internet applications." ACM SIGCOMM Computer Communication Review 31, no. 4 (2001): 149-160.

[15] Zhenjie, Liu. "Adaptive reorganization of database structures through dynamic vertical partitioning of relational tables." University of Wollongong Thesis Collection (2007): 33.

[16] Hammer, Michael, and Bahram Niamir. "A heuristic approach to attribute partitioning." In Proceedings of the 1979 ACM SIGMOD international conference on Management of data, pp. 93-101. ACM, 1979.

[17] Liu, Zhenjie, and Janusz R. Getta. "Optimization of query processing through constrained vertical partitioning of relational tables." (2006): 221.
[18] Rodríguez, Lisbeth, and Xiaoou Li. "A dynamic vertical partitioning approach for distributed database system." Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on. IEEE, 2011.
[19] Han, Jing, E. Haihong, Guan Le, and Jian Du. "Survey on NoSQL database." In Pervasive computing and applications (ICPCA), 2011 6th international conference on, pp. 363-366. IEEE, 2011.
[20] Corbett, James C., Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat et al. "Spanner: Google's globally distributed database." ACM Transactions on Computer Systems (TOCS) 31, no. 3 (2013): 8.
[21] Navathe, Shamkant, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. "Vertical partitioning algorithms for database design." ACM Transactions on Database Systems (TODS) 9, no. 4 (1984): 680-710.
[22] Curino, Carlo, Evan Jones, Yang Zhang, and Sam Madden. "Schism: a workload-driven approach to database replication and partitioning." Proceedings of the VLDB Endowment 3, no. 1-2 (2010): 48-57.
[23] Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." ACM SIGOPS operating systems review. Vol. 37. No. 5. ACM, 2003.
[24] Hunt, Patrick, et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." USENIX Annual Technical Conference. Vol. 8. 2010.
[25] Van Renesse, Robbert, et al. "Efficient reconciliation and flow control for anti-entropy protocols." proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware. ACM, 2008.
[26] Hayashibara, Naohiro, et al. "The φ accrual failure detector." Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on. IEEE, 2004.