Google Big Table

Xiao Chen
Betreuer: Marc-Oliver Pahl
Seminar Future Internet SS2010
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: Cx3606@gmx.de

ABSTRACT
Bigtable is a storage system for structured or semi-structured data [1]. Bigtable can be regarded as a distributed, non-relational Database from a system point of view (Bigtable is using a different data model than relational Databases). Bigtable scales to large data sets and provides a good performance of accessing the data. As of January 2008, there are more than sixty Google applications using the Bigtable as storage provider, such as Google Earth, Web-Indexing and Personalized Search. Bigtable fits different demands of those applications. It addresses problems which cannot be handled by standard relational databases. In this paper, we give a fundamental overview of Bigtable's design and implementation. We will describe the differences between Bigtable and relational database and focus on the different data models used by them.

Keywords
Bigtable, non-relational database, distribution, performance.

1. INTRODUCTION
The development of the Internet has introduced many new Internet based applications. The web-index and Google earth are for example used by millions of users from Internet. To manage these terabytes data (Google Earth has more than 70 terabyte data) becomes a challenge. To address the demands of those applications, Google has started the development of Bigtable in 2003. The Bigtable is designed to store structured or semi-structured data in nodes that are distributed over network.

The project is steadily growing since then. As of January 2008, there are over 600 Bigtable clusters at Google [2] and over sixty productive applications based on it. The design goal of the Bigtable mainly focuses on four aspects: high scalability, high availability, high performance, and wide applicability. There are some database models like "Parallel databases" [3] providing speed up and scale up of relational database queries, but Bigtable distinguishes itself from those models: it is not a relational database. It provides a different interface as those relational database models.

Bigtable shares many database strategies: Data scan, Data Storage and Data access. Unlike a relational database which stores a fixed schema in database server, the data logic is embedded in the client code: the client code controls dynamically how the data structure is.

Bigtable relies very much on a so called mapreduce job to guarantee the design goal of a high performance of the distribution. Mapreduce [4] is a framework for processing and generating large data sets. Bigtable is used as input or output source for Mapreduce jobs. Mapreduce Job provides a very fast transformation for the data of Bigtable to hundreds of nodes across the network.

In Section two, we firstly introduce an application example which is using Bigtable as storage provider. We are going to see the potential requirements of this application and why the standard Relational Database Management System (RDBMS) cannot be used for this application. We will explain briefly how Bigtable solves those demands from a high level design strategic perspective. Section three contains information of differences between the RDBMS and the Google Bigtable. We will introduce the data model used by the two Database models. This will give a further explanation why Bigtable is more suitable to store large datasets in a distributed way. Section four will provide an overview of the building blocks of Bigtable. Section five introduces the basic implementations. In section six we describe some refinements Bigtable is using to archive the design goals. In Section seven we will give a short overview of the client API. Section eight presents the entire architecture of Bigtable and our conclusions of the design principles.

2. APPLICATION EXAMPLE: GOOGLE EARTH
Google Earth is one of the productive applications which are using Bigtable as storage provider. It offers maps and satellite images of varying resolution of the Earth's surface. Users can navigate through the earth surface, calculate a route distance, execute complex or pinpointed regional searches or draw their own routes. In following section, we introduce some fundamentals of the implementations. We will see why Bigtable can address the requirements of Google Earth better than a standard relational Database.

Google Earth is using one table to preprocess raw data, and several other tables for serving the client data. During preprocessing, the raw imagery is cleaned and consolidated into serving data (the final data used by the application). The preprocessing table stores the raw imagery. It contains up to 70 terabytes data and therefore cannot be maintained in the main memory. It is served from the disk. The imagery was already be consolidated efficiently, for this reason Bigtable compression is disabled. The details for Bigtable's compression methods can be found on section 6.1 compression.

The size of the preprocessing table is the first reason why we cannot use RDBMS to store the data, the 70 terabytes data cannot be stored as one table hosts in a single machine.

The serving system of Google Earth is using a single table to store the index data. Although the index table is relative small (500GB), if we use one machine to host the table we still have to move the data to hard disk. For performance considerations, we cannot implement it, because the index table must serve tens of thousands of queries per second per datacenter with low latency. With the data stored on disk, we would have no chance to fulfill the requirements with the current hardware technology. We need a solution to distribute the data into multiple nodes.

A major characteristic of Bigtable is its scalability. Relational Database also scales but only in single node. When the hardware capacity of the single node is reached, the load needs to be distributed to other nodes. Some Databases like Oracle provide services like replication jobs to address scale loads out of a single machine. For an application like Google Earth which has a massive workload, it will require hundreds or thousands of nodes' capacity to store the data. Standard replication jobs cannot be used in this kind of situation. RDBMS is more suitable for applications which are hosted on one single node but Bigtable is designed for data distribution of a large scale into hundreds or thousands of machines over network. Mapreduce Job [4] is always used for Bigtable applications to distribute and to process the data to nodes and within network.

By speaking of the scalability requirements, another consideration is the flexibility. This can also become a problem of managing the system if we only have RDBMS located on a single node. Taking the index table used by Google Earth as example, when the server load or the size of the index table becomes double and the hardware capacity of this single node is reached we cannot upgrade the hardware on the single node as fast as the speed of the change. Bigtable allows managing the data in a flexible way to add or remove a node from the distribution cluster. More details about how Bigtable manages the tablets assignment can be found in section 5.3.

3. DIFFERENT DATA MODEL USED BY BIGTABLE AND RDBMS

The example of Google Earth presents the motivations of using Bigtable. We will explain the different Data Models used by Bigtable and RDBMS. This will on one hand demonstrate why Bigtable is not a relational database system. On the other hand, it will give explanations why Bigtable is more suitable for applications which require distributed storage.

A relational database is a collection of tables (entities). Each table contains a set of columns and rows. The tables may have constrains to each other and relationships between them.

Figure 1 shows a typical Data model used by RDBMS. The column Id from entity "Functionary" is used as foreign key which is referenced to column "idSupport" of entity "Support". The column "idAttendee" of entity "Attendee" is referenced to column "AttendeeId" of Entity "Support". The relationship (Data logical) between "Functionary" and "Attendee" is thus kept in entity "Support".

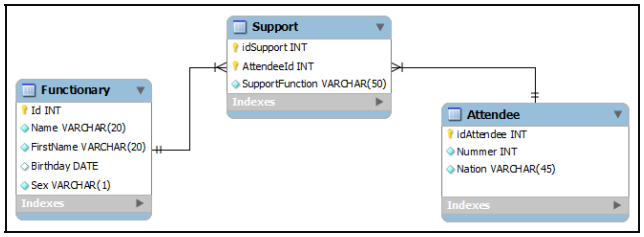

Figure 1. relational Data Model: functionary has constrains with Attendee via Support entity.

The RDBMS model exists since almost 30 years. When it was developed, the RDBMS was not widely used due to hardware limitations. Even a simple select statement may contain hundreds of potential executing paths which the query optimizer needs to calculate at runtime. Today, the hardware can satisfy the demands of RDBMS, so the relational Database has become a dominant choice for common applications. They are easier to understand and to be used - although it still provides less efficiency compared to the legacy hierarchically database. Almost all databases we are using now are RDBMS; typical examples are Oracle, MS Sql and DB2.

Comparing to RDBM, which stores the data logical within table itself, Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp. It can thus be seen as a 3D table in form of row*column*timestamp. Each value in the map is an un-interpreted array of bytes. The data model used by Bigtable can be concluded as follow formula:

(row: String, Column: String, time: Int64) → String

The row keys are a user-defined string. The data is maintained in Bigtable in a lexicographic order by the row key.

Column key syntax is written in the form of "Family:optional_qualifier". The Column keys are grouped together into the Column families. Column families can be regarded as categories of its column. The data belongs to the same Column family is compressed together with the column information. The Column families are the basic unit of accessing the data. They can also have attributes/rules that apply to their cells, such as "keep n time entries" or "keep entries less than n days old".

The timestamp is used to version the data, so we can track the changes of the data over the time. The size of the timestamp is a 64bit integer. Each column family can have different number of versions. For example, a data string in Bigtable has two column families: content and anchor. Content column family contains the page of content, and the anchor column family contains text of anchors which references to the page. The two column families can have different number of versions based on application's requirement. The content column family can keep the latest 3 updates while the anchor column family only keeps the latest update.

The timestamp can be used together with the rules added to Column to manage the lifecycle of the date. The old timestamps could be removed by a garbage-collection.

After this short introduction to the datamodel used by Bigtable, we can now see that its data model does not equal the Data Model used by a relational database. Bigtable characterizes itself as a database management system which is commonly called a key/value Database [6], the name is not official, and some documentation also refer to this kind of database as distributed database or distributed hashtable.

4. BUILDING BLOCKS

Bigtable cluster contains one or several tables. Each table consists of set of tablets by the range of the row key. The tablets are usually around 100-200MB and can be distributed into different nodes across the network. Each node(tablet servers) saves about 10~1000 tablets wtihin its Google File System(GFS) [7]. Figure 2 shows a simpfied overview of the structure of the Bigtable implementation and the major building blocks are used there: Google File System (GFS), SSTable (sorted string table) and Chubby.

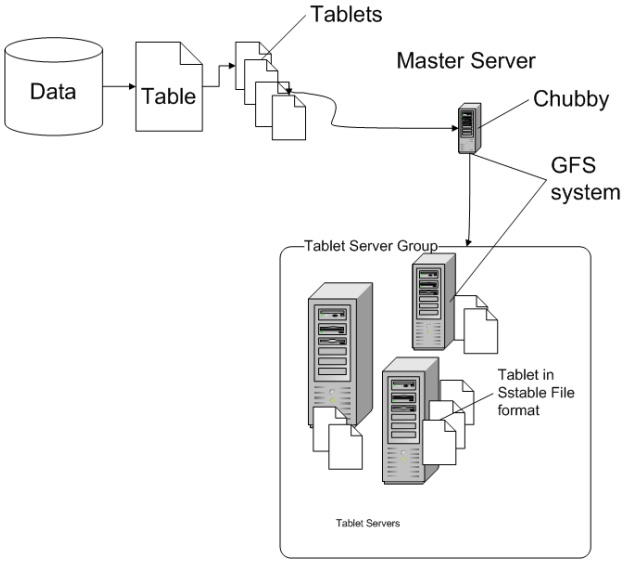

Figure 2 Building Blocks: Chubby, GFS, SStable

GFS is a File system used by Bigtable to store the logs and files. A Bigtable cluster usually runs a shared pool of the distributed nodes. Bigtable also requires a cluster management system (CMS). The CMS is used to schedule jobs, manage resource on shared machines, replicate jobs of failure machines and monitor the machine status.

SSTable stands for "sorted string Table", this is a immutable file format which Google internally used to save the Tablets. The paper [5] of the Bigtable provides follow information for SSTable:

"An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. Operations are provided to look up the value associated with a specified and to iterate over all key/value pairs in a specified key range."

Internally, the SSTable contains a set of blocks which is 64KB. This size can be configurable by application requirements. On the end of the SSTable an index will be generated and it will be loaded into memory when an SSTable is opened. In this way, a lookup can be done using single disk seek: we just need to find block using the index stored in memory and then read the block from the disk. Figure 3 summarize the implementations of the SSTable:

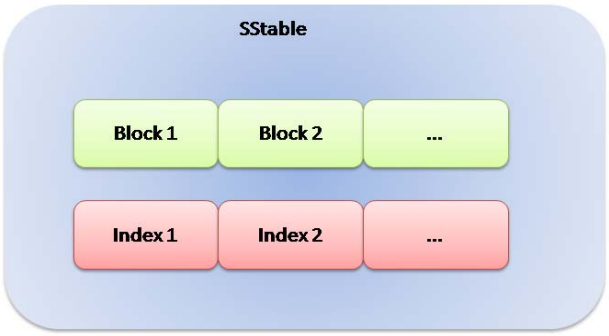

Figure 3 SSTable schema: index of each block will be loaded in memory. So the lookup to the actual data only needs one disk seek.

Alternatively, an SSTable can be completely copied to the main memory to avoid read from disk.

Bigtable depends on a highly-available and persistent distributed lock service called Chubby [8]. It is used to keep track of tablet servers in Bigtable. Chubby itself is a cluster service that maintains five active replicas, one of which is the master. The service is running when majority of the replicas are running and can communicate with each other. As explained in the document of Bigtable [5], Chubby provides a namespace that consists of directories and small files. Each directory or file can be used as a lock, and reads and writes to a file are atomic. The Chubby files are cached consistently in the Chubby client library. Chubby allows the client to take a local lock, optionally with the metadata which is associated with it. When a client session has been expired, Chubby will revoke all locks that it has. The client can renew the lock by sending "keep alive" messages to Chubby.

Chubby can be seen as the global lock repository of Bigtable. How Bigtable is using the Chubby can be seen by the section 5.3 tablet assignment.

5. IMPLEMENTATION

5.1 Major Components

The Bigtable implementation contains three major components: A Client library, one Master Server and many Tablet Servers.

The master server is responsible for assigning tablets to the tablet server. It detects the additional and expiration of the tablet server in order to balance the tablet server load. It also does the garbage collection of the files on the GFS. Furthermore, when the schema of the rows and column families need to be changed, the Master Server also manages those changes.

Tablet server is designed to manage the set of tablets. It handles the read and write requests to the tablet. When a tablet is growing too large, the Tablet server also splits it into small tablets for the

future processing. Tablet servers can be added or removed from a Bigtable cluster based on the current workload. This process is managed by master server.

Although there is a one single master server existing in the cluster, since the clients do not rely on master server for the tablet location information, the load of the master server is very low. The client communicates with the tablet server directly to read or write data from a tablet.

5.2 Tablet Location

Bigtable is using a three-level hierarchy to store the tablet location analogous to that of a B+ [9] tree: Root tables, metatables and usertables (Figure 4)

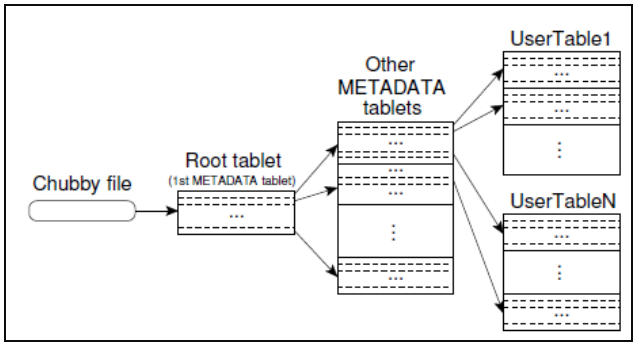

Figure 4 Table location hierarchies based on Bigtable paper [5]

Chubby stores a file which contains location information to a root tablet. The root tablet contains location information to other metadata tablets in a special Metadata tablet. This Special Metadata tablet will never be split – to ensure the location hierarchy is never expanded more than three levels.

Each client library contains a cache to the location information. If the cache is empty or if the client detects that the cache information is not correct, the client will move up the hierarchy to retrieve the location recursively. In worst case, a location look up through the three-level hierarchy may require six network round trips including a lookup in the Chubby file.

Bigtable stores secondary information in the metadata table like logs (when a server begins serving it), such information is helpful for troubleshooting or performance analytics.

5.3 Tablet Assignment

A tablet server contains a set of tablets. Each tablet can be assigned to one tablet server at one time. The master server is used to keep tracking the set of live tablet servers, managing the current assignments of tablets to tablet servers. When a tablet becomes unassigned, the master server will verify firstly if there is enough space on the live tablet server, it will then assign the unassigned tablet to this tablet server.

The Chubby service which we mentioned in section 4 is used by master server to keep track of tablet servers.

When a tablet server starts up, it creates, and gets an exclusive lock on a unique-named file in the Chubby directory [5]. This directory is monitored by the master server, so the master can discover the newly arrived tablet servers.

When the tablet server loses the exclusive lock on the directory, it stops serving. The tablet server will reacquire the exclusive lock of the file as long as it still exists in the directory. If the file does not exist, the tablet server will kill itself.

When the tablet server terminates itself, the tablets which are associated within the tablet server will be unassigned, because the tablet server will attempt to release its lock. The Master can then reassign those unassigned tablets as soon as possible.

The master asks the lock status of the tablet server periodically. If the tablet server replies with a loss of the lock status, or the master does not get a reply from a certain tablet server after several attempts, the master server will try to acquire an exclusive lock on the file itself. If the file can be locked, it implies that Chubby is alive and the tablet server which locked this file died. The master will ensure that this tablet server can't serve any data again by deleting the server file.

Once the server file is deleted, those tablets which are previously assigned to this tablet server become unassigned.

As mentioned in section 4, Google uses the CMS system to manage the clustered nodes. When a master server is started by the CMS, it needs to be informed of current tablet assignment. In the following step the assignment information is collected:

1. Initialize a unique master lock in Chubby to prevent other concurrent master server from initializing.
2. Scan the server file directory to recognize the live tablet servers.
3. Tell the master server to connect to each live tablet server to scan the tablets which are assigned to the tablet server.
4. The master scans the metadata table to learn the set of unassigned tablets.

5.4 Implementation Memtable

Memtable is the in-memory representation of the most recent updates of a tablet. It contains the recently committed logs which are stored in memory in a sorted buffer.

The memtable will not increase infinitely. When it reaches its threshold (depends on the main memory size), the current memtable will be converted to an SSTable and moved into the GFS, a new memtable will also be created. This process is called Compaction. Those SSTables act as snapshots of this server so they can be used for recovery after a failure.

When a write option arrives, the tablet server checks the well-formedness and the authorization to see if an option is permitted and stores it into the commit log. After the write has been committed, the content is inserted into the memtable.

When a read option arrives, the tablet server will also check if the option is well formed and if the sender of the option is authorized. If it is a valid operation, the operation is executed in a merged view of a sequence of the SSTable and the Memtable. Since the SSTable is sorted in a lexicographically way it is easy to form the merged view.

When those tablets are splitting or getting merged, it does not lock the read/write operations.

Figure 5 is made to represent how read and write is done and procedure of the Memtable.

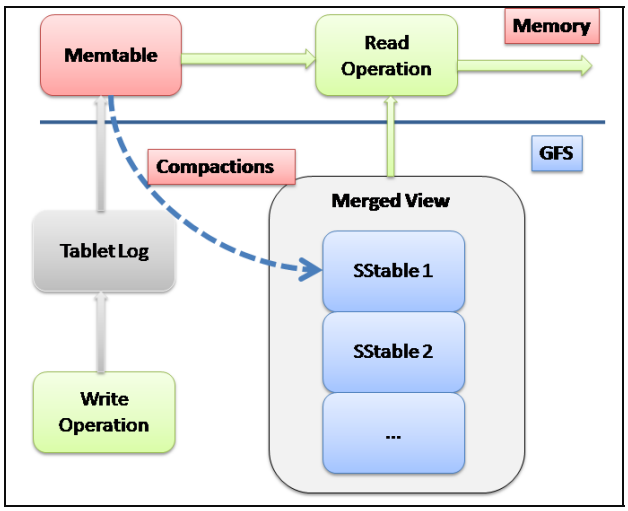

Figure 5 read write process, the read process is performed on a merged view of the SSTable. The write operation is written into the Tablet Log. When Memtable increases its size, it will be converted to SSTables and moved to GFS

6. REFINEMENT USED BY BIGTABLE

6.1 Locality groups

Locality Groups are user to group parts of data together which have similar user criteria. For example, the metadata of a webpage can be grouped together as one locality group, while the content of the webpage is grouped as another locality group

6.2 Compression

The Bigtable implementation relies on a heavy use of the compression. Clients can specify whether or not SSTables for a locality group are compressed or not. Client also specifies which compression schema is to be used.

A typical two-pass compression schema is used by many clients. The first pass uses Bentley and McIlroy's scheme [10], which is designed for compressing very long strings. The second pass looks into the small 16kb window for repetitions of the data. The first and second pass is done in a very quick way, the encoding cost is 100-200MB/s and the decoding cost 400-1000MB/s. Even those two schemata are chosen for a quick decoding and encoding process. In t practice they provide a high rate as 10 to 1 of the compression.

6.3 Merging unbounded SSTables

One optimization used in Bigtable is merging of unbounded SSTables. A single SSTable is merged from SSTables for a given tablet periodically. This single SSTable contains also a new set of updates and index. This will prevent that the read option loading every data from this small piece of SSTable and access the GFS many times.

6.4 Caching

The caching is mainly used to improve the read performance. There are two levels of caching used by Bigtable: scan cache and block cache. Scan cache is a high level cache which caches the key-value pairs returned by SSTables. It is most useful for applications which are reading data repeatedly. The Block Cache is a low level cache. It is useful for applications which read Data close to data they recently read.

6.5 Bloom filter

A very important problem by using Bigtable is the access of the SSTable files. As mentioned in section 5, the SSTable is not always kept in memory, the user read operation may need many accesses in the GFS (located in the hardware layer) to load the state of the SSTable files. The paper of Bigtable [5] explains that they reduce the number of accesses by allowing clients to specify that Bloom filters [11] should be created for SSTables in a particular locality group. Bloom filter is an algorithm which is used to verify if a data is in the membership of the set. In the implementations of Bigtable, the bloom filter is kept in memory in order to probabilistic if a data exists in a given row/column pair. The memory usage to store the bloom filter is very small, but this technique drastically reduces the access of the data in disk.

7. API

In the relational database, the data will be updated, inserted, deleted using SQL Statements. Bigtable does not support SQL (it supports a Google designed language called Sawzall [12] to filter the data). Client applications can write or delete data, lookup values from rows and column families within Bigtable using API Method calls. The application and data integrity logic is thus contained in the application code (not like the relational data, the embedded logic is stored in the Data model with triggers, stored procedure, etc.).

As mentioned in section 2, Bigtable is storing the data in form of row*column*timestamp, the column family is the category of the column. Figure 6 shows a simple API code written in C++ to modify the data stores in Bigtable.

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

Figure 6 Open a table and write a new anchor, which is a column family, and then delete the old anchor. [5]

8. CONCLUSION

Taking account of the above, Figure 7 shows a simplified overview of the Bigtable's Architecture.

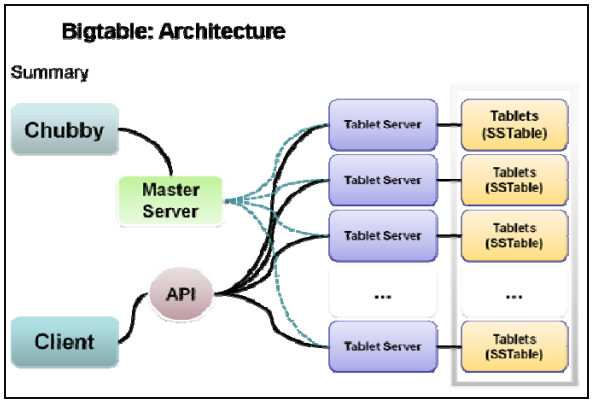

Figure 7 An overview of the Bigtable Architecture

Bigtable varies from traditional relational database on its data model. Bigtable tends to be used by applications like Google Earth, which require a large storage volume. Legacy network database or relational database can not address the requirements by those kinds of applications to distribute data over thousands of hosts. A relational database system is more powerful and is still a dominant choice for those applications which require a storage that is hosted by several hosts.

On the other hand, large distributed systems are more vulnerable to many types of failures: memory and network corruptions, large clock skew and Chubby service failures. All those problems could cause Bigtable implementations to fail. Some of those problems are addressed by changing various protocols used by Bigtable, but the implementations still need further refinements.

Here are some design principles which can be extracted from the Bigtable implementation:

- Use single master server for a quick, simple management of distribution.
- Use refinements technique to avoid accessing the disk directly. The latencies caused by read operations will be more expensive as the network round trips.
- Replicate the storage to handle the failure of cluster node.
- Avoid replicating the functionalities: it is more expensive to keep the server in sync as to replace a failed server. So we should not replicate the functionalities in different server.
- Make a high available rate of the communication/lock service. Chubby is an example: If this service fails, most Bigtable operations will stop working.

9. REFERENCES

[1] **Buneman, Peter.** *" Tutorial on semi-structured data. ",* 1997.

[2] **Wilson Hsieh, Jayant Madhavan,Rob Pike.** *" Data management projects at Google. ",* Chicago, IL, USA : ISBN:1-59593-434-0 , 2006 . ACM SIGMOD international conference on Management of data. S. 36.

[3] **David Dewitt, Jim Gary.** *" Parallel database systems: the future of high performance database systems. ",* 6, New York, NY, USA : ACM, 1992, Bd. 35. ISSN:0001-0782 .

[4] **Jeffrey Dean, Sanjay Ghemawat.** *" MapReduce: Simplified Data Processing on Large Clusters. ",* 2004. OSDI '04 Technical Program. S. 1.

[5] **Fay Chang, 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. ",* WA : s.n., 2006. OSDI'06: Seventh Symposium on Operating System Design and Implementation. S. 1.

[6] **Bain, Tony. "** http://www.readwriteweb.com/enterprise/2009/02/is-the-relational-database-doomed.php. ", *" ReadWrite. ",* [Online] 12. February 2009.

[7] **Sanjay Ghemawat, Howard Gobioff,Shun-Tak Leung.** *" The Google file system. ",* New York,USA : ACM, 2003 .

[8] **Burrows, Mike.** *" The Chubby lock service for loosely-coupled distributed systems. ",* Berkeley, CA, USA : USENIX Association, 2006. ISBN:1-931971-47-1.

[9] **COMER, Douglas.** *" Ubiquitous B-tree. ",* 2, New York : ACM Computing Surveys, 1979, Bd. 11. ISSN:0360-0300.

[10] **BENTLEY, J. L., AND MCILROY.** *" Data Compression Using Long Common Strings. ",* 1999, In Data Compression, S. 287-295.

[11] **Bloom, Burton H.** *" Space/time trade-offs in hash coding with allowable errors. ",* 1970, Commun. ACM, S. 422-426.

[12] **Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan.** *" Interpreting the data: Parallel analysis with Sawzall . ",* 2005, Scientific Programming, S. 227–298.