Enterprise Computing: Exercise 5 – BigTable, MongoDB, Cassandra

Marco Peise, Stefan Tai

Wirtschaftsinformatik –
Information Systems Engineering

Agenda

Lecture BigTable

Exercise 5

- Cassandra
- MongoDB

Agenda

1. Intro Cloud Storage

2. Systems in detail
   a) Dynamo
   b) GFS, **BigTable**

3. Other systems
   c) Cassandra
   d) MongoDB

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

BigTable – Motivation

GFS only provides means to store unstructured data – however, some applications require a certain amount of structure

GFS was made for large files – some applications need to store small data items

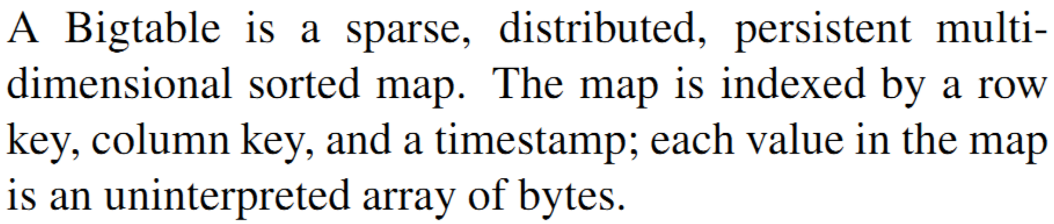
All the positive aspects (scalability, high availability, high performance etc.) should be kept

Reference:
F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, R. Gruber: Bigtable: A distributed storage system for structured data, OSDI, 2006.

What is Bigtable?

Building on GFS Bigtable provides additional structure, query semantics and support for small files

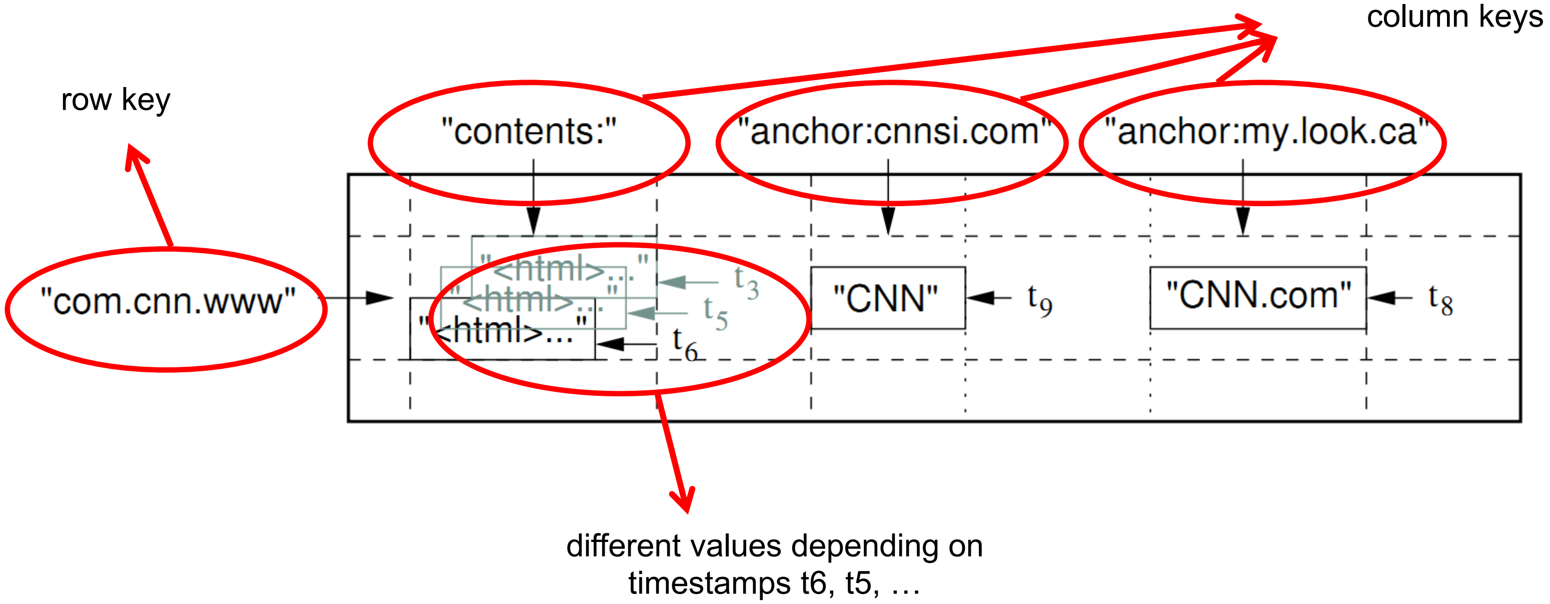
Chang et al. phrase it as:


A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

Used by more than 60 Google projects already in 2006, including MapReduce, Earth, Analytics and Orkut

Data Model

(row:string, column:string, time:int64) → value:string



Data Model

Zoo Example

row key,	col. key,	timestamp	
(zebras,	length,	2006)	--> 7 ft
(zebras,	weight,	2007)	--> 600 lbs
(zebras,	weight,	2006)	--> 620 lbs

Each key is sorted in Lexicographic order

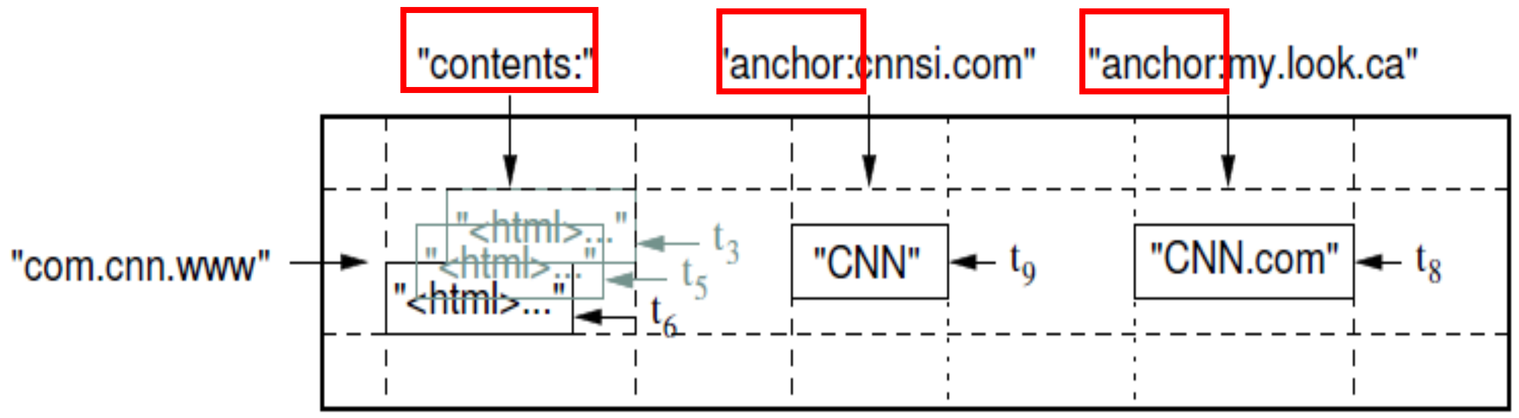
Data Model

Zoo Example

row key	col. key	timestamp	
(zebras,	length,	2006)	--> 7 ft
(zebras,	weight,	2007)	--> 600 lbs
(zebras,	weight,	2006)	--> 620 lbs

Timestamp ordering is defined as "most recent appears first"

Data Model



Column family

family: qualifier

Structure and Query model

A table contains a number of rows and is broken down into tablets which each contain a subset of the rows

Tablets are the unit of distribution and load balancing

Rows are sorted lexicographically by row key

- → subsequent row keys are within a tablet
- → Allows efficient range queries for small numbers of rows

Operations: Read, write and delete items + batch writes and Sawzall scripts running on (!) tablet server

Supports single-row transactions

API

Metadata operations

- Create and delete tables, column families, change metadata

Writes (atomic)

- Set(): write cells in a row
- DeleteCells(): delete cells in a row
- DeleteRow(): delete all cells in a row

Reads

- Scanner: read arbitrary cells in BigTable
  - Each row read is atomic
  - Can restrict returned rows to a particular range
  - Can ask for just data from one row, all rows, a subset of rows, etc.
  - Can ask for all columns, just certainly column families, or specific columns

API

Write

```
// 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);
```

Read

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s\n",
         scanner.RowName(),
         stream->ColumnName(),
         stream->MicroTimestamp(),
         stream->Value());
}
```

Elements

A client-side library for access to Bigtable

A single master server

- assigning tablets to tablet servers,
- detecting the addition and expiration of tablet servers,
- balancing tablet-server load,
- garbage collection of files in GFS,
- handles schema changes such as table and column family creations

A number of tablet servers

- Each server stores a number of tablets (10-1000) at approx. 100-200MB each
- Handles read and write requests
- Splits tablets that have grown too large

Tablet Server

Tablet server startup

- It creates and acquires an exclusive lock on a uniquely named file on Chubby.
- Master monitors this directory to discover tablet servers.

Tablet server stops serving tablets

- If it loses its exclusive lock.
- Tries to reacquire the lock on its file as long as the file still exists.
- If file no longer exists, the tablet server will never be able to serve again.

Master Server

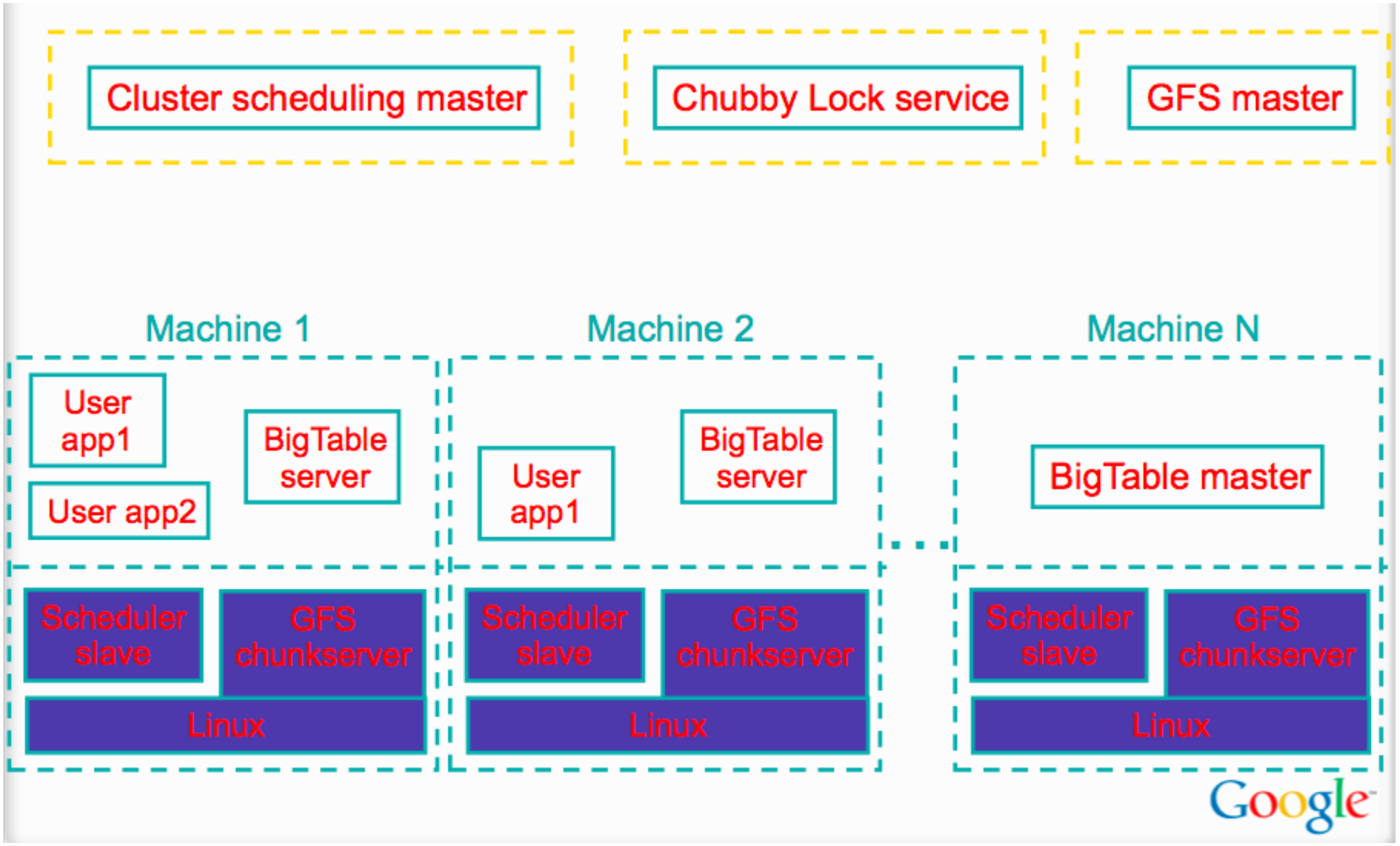
Master server startup

- Grabs unique master lock in Chubby.
- Scans the tablet server directory in Chubby.
- Communicates with every live tablet server
- Scans METADATA table to learn set of tablets.

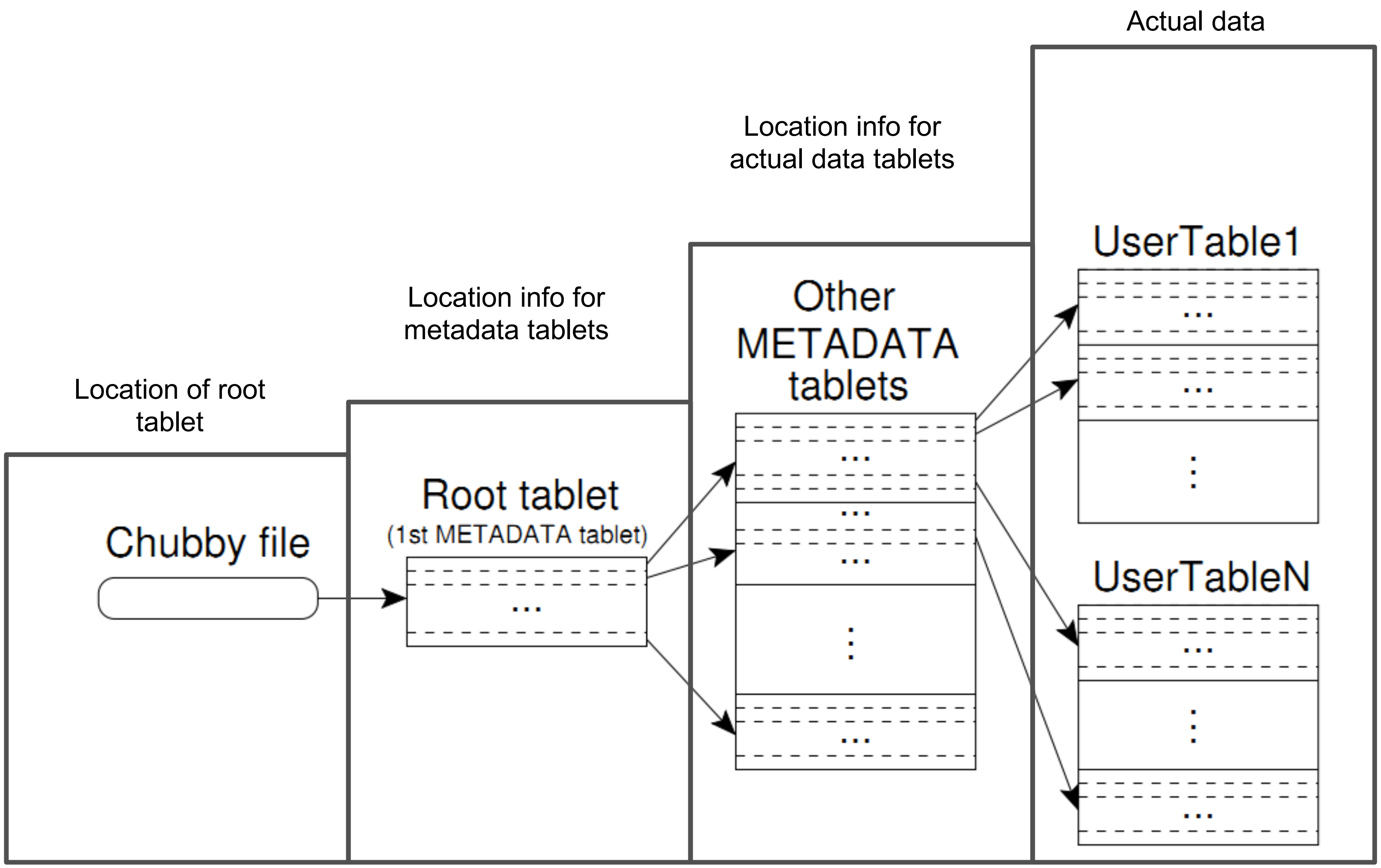
Master is responsible for finding when tablet server is no longer serving its tablets and reassigning those tablets as soon as possible.

- Periodically asks each tablet server for the status of its lock
- If no reply, master tries to acquire the lock itself
- If successful to acquire lock, then tablet server is either dead or having network trouble

Building Blocks

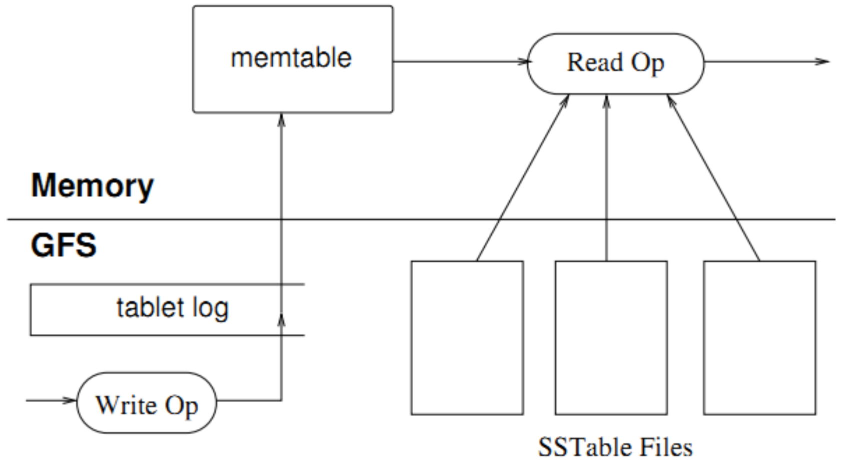


Tablet location



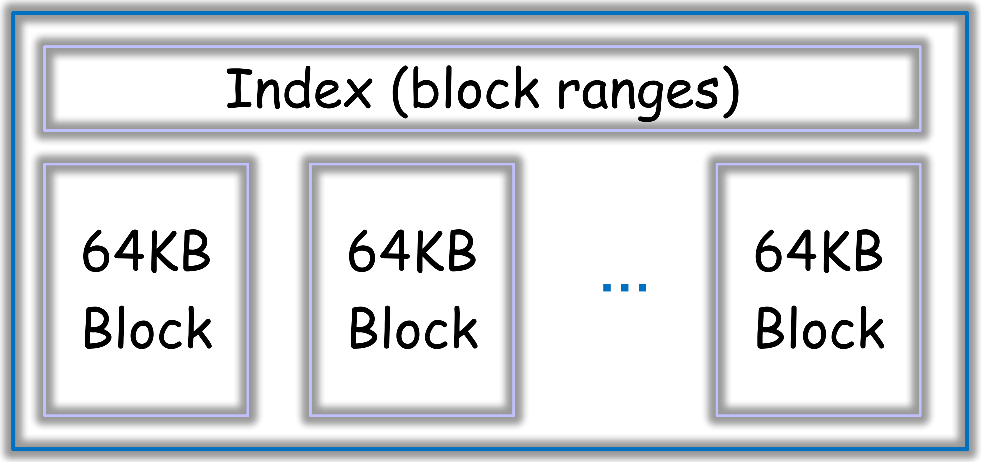
Local persistence

Tablet servers store updates in commit logs written in so-called SSTables
Fresh updates are kept in memory (memtable), old updates are stored in GFS
Minor compactions flush memtable into new SSTable
Major compaction merge SSTables into just one new SSTable



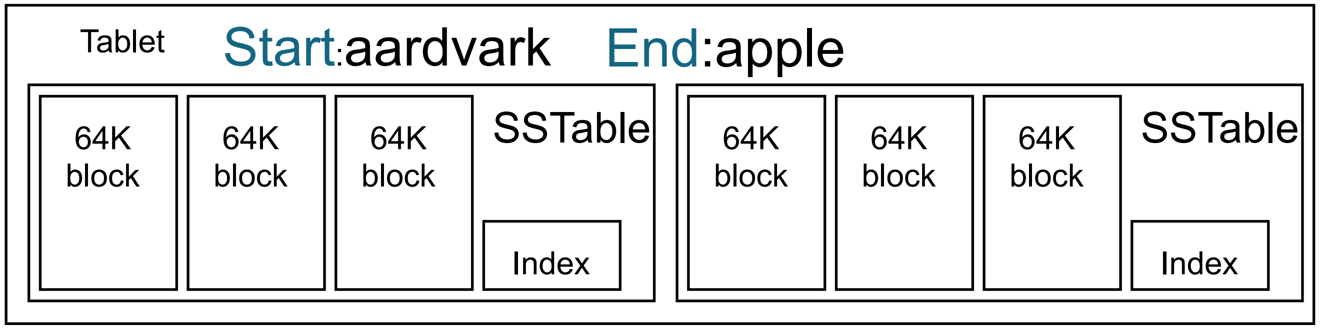
SSTable

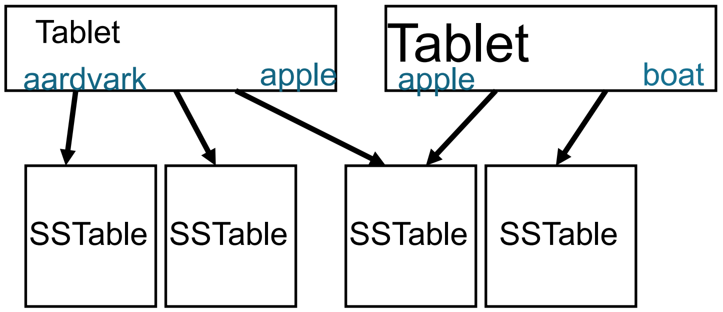
- An SSTable provides a persistent , ordered immutable map from keys to values
- Each SSTable contains a sequence of blocks
- A block index (stored at the end of SSTable) is used to locate blocks
- The index is loaded into memory when the SSTable is open
- One disk access to get the block



Tablet

- Contains some range of rows of the table
- Built out of multiple SSTables





Summary

Bigtable enhances GFS with a structured data model and more sophisticated query functionality

It adds support for small files while still maintaining system properties like

- High availability
- High throughput
- Low latency
- Slightly sub-linear scalability

It abstracts the user from GFS hassles with asserting that data is consistent and defined

Exercise 5

Task 1 Cassandra a) - Keyspaces

Does the following CQL3 statement create a new keyspace that places one replica in each of two data centers?

```
cqlsh> CREATE KEYSPACE <ksname>
  WITH REPLICATION = {
    'class':'NetworkTopologyStrategy',
    'DC1':1,
    'DC2':1
  };
```

Task 1 b) - Compound primary keys

Which statement creates a table "messages" in the keyspace "msgplatform" that is partitioned by "participant_id" across multiple servers and locally clustered by "posted_on" date and “posted_location” string?

Task 1 c) - Add a collection to a table

You want to alter an existing "customers" table schema with an additional column that can contain sets of "interests" (text strings). After you have altered the table, a query like this should work:

```
cqlsh:msgplatform> UPDATE customers SET interests = interests + {'climbing','baking'} WHERE id = 202;
```

Please complete the following uncomplete statement using correct SQL syntax:

```
cqlsh:msgplatform> ALTER TABLE customers ADD ______
```

Task 1 d) - TTL

Is it true or false? The following statement inserts an expiring column named "pw_reset_token" with value "fgh-lmn-456" into the "customers" table. The column "pw_reset_token" will be deleted automatically after 6000 seconds but the customer row with id 315 and name "karlmann" will still be present in the table after 6000 seconds.

```
cqlsh:msgplatform> INSERT INTO customers(id,
  name,
  pw_reset_token)
  VALUES (315, ’karlmann', ’fgh-lmn-456')
  USING TTL 12000;
```

Task 1 e) - Counter columns

You have created a table named ”referendum" with two counter columns like this:

```
cqlsh:twotter> CREATE TABLE referendum (
  participant_id int,
  msg_created_time timeuuid,
  msg_created_location string,
  voteinfavor counter,
  voteoppose counter,
  PRIMARY KEY (participant, msg_created_time,
  msg_created_location)
  );
```

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Task 1 e) - Counter columns

Complete the following query that increments the number of in favor votes for a referendum which are created in "germany", between the first and second of January 2016 (midnight-midnight) and from the participant with the number "519":

```
cqlsh:msgplatform> UPDATE referendum SET ___
```

Task 1 f) - Lightweight transactions with Compare-and-Set (CAS)

Write two CQL queries that implement the following feature:

1. A user named Catarina (user id: “689”) wants to reset her password by creating an expiring token ("pw_reset_token") column in her user row in the “customers” table.
2. She gets a message with a link to a site where to reset her password. On that site she can enter a new password only if the expiring token (encoded as parameter into the link in her message) matches the expiring token in her user row and if the token has not expired, yet.

Task 1 g) True or False?

Only one node in a P2P data cluster (such as a Cassandra cluster) maintains a sliding window of inter-arrival times of Gossip messages.

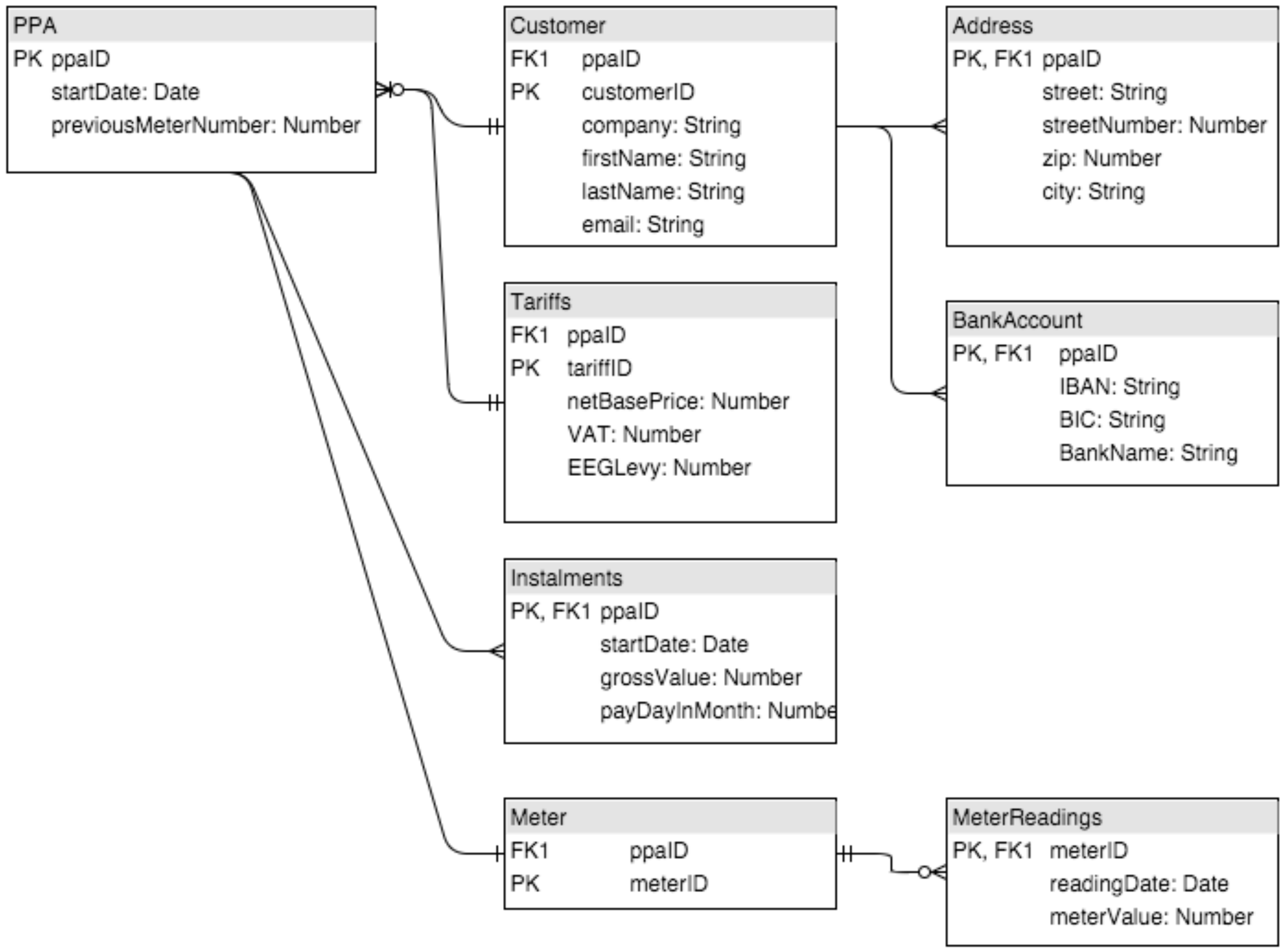
Task 2 – MongoDB – Modeling a Scenario

Create a DB Model to invoice Energy Customers in Germany.

Given Information:

- PPA, Customers, Customer Contact Information, Bank Information, Tariffs, Instalments, Meter Information, Meter Readings

Task 2 – MongoDB – Modeling a Scenario



Task 3 – MongoDB – Modeling a Scenario

Task: Complete a list of commands to obtain a JSON Structure as provided in the exercise sheet.

1) Download and install MongoDB from: https://www.mongodb.org/downloads

2) Carefully observe the given JSON Structure from the exercise sheet.

3) Execute the given commands in MongoShell

4) Complete the set of Commands by using only „save“ or „update“ operations in MongoDB

Task 3 – MongoDB – Modeling a Scenario

```
{
  _id: <ObjectId01>,
  startDate: „<YYYY-mm-dd>“,
  previousMeterNumber: 123456789,
  instalments: [{
                         startDate: „<YYYY-mm-dd>“,
                         grossValue: 39
                       },
                       {
                         startDate: „<YYYY-mm-dd>“,
                         grossValue: 40
                       }
  ],
  meter: [{
               readingDate: „<YYYY-mm-dd>“,
               meterValue: 20
             },
             {
              readingDate: „<YYYY-mm-dd>“,
              meterValue: 40
             },
             {
              readingDate: „<YYYY-mm-dd>“,
              meterValue: 60
             }
  ],


}
```

```
customer: {
      company: „Smith Inc.“,
      firstName: „John“,
      lastName: „Smith“,
      email: „john@smith.com“,
      address: [{
              street: „UperStreet“,
              streetNumber: 15,
              zip: 10411,
              city: Berlin
            },
            {
              street: „LowerStreet“,
              streetNumber: 19,
              zip: 10413,
              city: Berlin
            }],
      bankAccounts: [
                  {
                   IBAN: 123456789,
                   BIC: 112233445,
                   bankName: BankOfEngland,
                  },
                  {
                   IBAN: 123456799,
                   BIC: 552233445,
                   bankName: BankOfScottland,
                  }
         ]
},
tariff: {
      netBasePrice: 4.99,
      VAT: 19,
      EEGLevy: 6.354
}
```

Task 3 – MongoDB – Modeling a Scenario

Here are the commands which should have been executed:

```
1.   db.testing.save({startDate:new Date()});
2.   ...
3.   db.testing.update({„_id“:ObjectId(„xxx“)},{„$set“: {„meter“: []}});
4.   ...
...
9.   ...
10.  db.testing.update({„_id“:ObjectId(„xxx“)},{„$set“: {„customer“: {company: „Smith Inc.“,email: „john@smith.com“}}});
11.  ...
12.  ...
13.  ...
14.  db.testing.update({„_id“:ObjectId(„xxx“)},{„$set“: {„customer.bankAccounts“: []}});
15.  ...
16.  ...
17.  db.testing.update({„_id“:ObjectId(„xxx“)},{„$set“: {„tariff“: {netBasePrice: 4.99,VAT:19,EEGLevy:6.354}}});
```