Proposta progetto DHT

Alessandro Di Stefano - Marco Grassia

Consegna: Home Work 6

SCANNER#2: programma Java che esegue uno script per il recupero di informazioni relative alle performance (Cpu/Mem, disk I/O, network I/O)

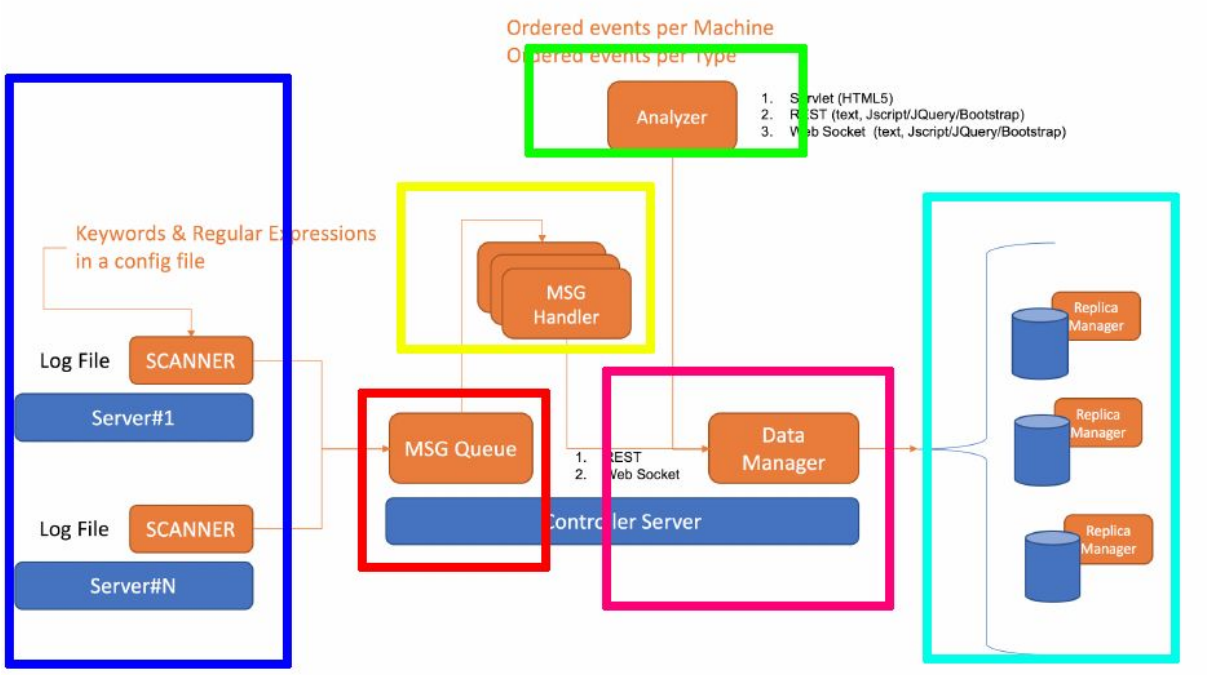
MSG QUEUE: riceve i dati (asynch) dagli SCANNER e li invia ai vari MSG Handler
● MSG Handler Topic-based: salva sul db usando i servizi del data manager (synch).

IL DATA MANAGER, sviluppato a componenti EJB, rappresenta l'interfaccia Read (per Analyser) / Write (MSG Handler) per i database.

DB distribuito su una DHT. prevedere join, fault e leave e strategia di fault tolerance
ANALYSER "acts as CONTROLLER".

Mette a disposizione all'utente finale query relative ai dati memorizzati.
es.
- Eventi ordinati nel tempo per tutte le macchine
- Eventi specifici ordinati nel tempo per tutte le macchine
- Eventi ordinati nel tempo per una singola macchina
- Eventi specifici ordinati nel tempo per una singola macchina
- (Tempo, evento, macchina come parametri principali)
- Dati su info per macchina
- Dati su info per l'intero sistema
- Dati su specifiche finestre temporali

Architettura proposta



Descrizione generale

Ognuno dei rettangoli rappresenta un gruppo di Docker Container, gestiti con *docker-compose*.
È garantita, dunque, la possibilità di replicare i servizi che lo necessitano, e nel complesso un mini cloud data-center virtuale, nel quale coesistono il meccanismo di monitoring e i nodi "utente".
In una fase successiva dello sviluppo, potrebbe essere eseguito il deploy dell'ambiente (software-defined) e del software di monitoring implementato in un qualunque sistema cloud-oriented.

Server & Scanner service

I server analizzati dal monitor sono rappresentati dai blocchi racchiusi nel rettangolo blu, e possono, virtualmente, fornire servizi differenti.

Un *daemon* presente in ogni server, chiamato *scanner*, è responsabile dell'invio di statistiche in merito all'utilizzo delle risorse locali.

Specifiche tecniche dello scanner:

- Linguaggio: script (POSIX bash\sh)
- Funzionamento generale: esecuzione periodicizzata tramite Cron ed invio delle letture alla message queue tramite API apposite
- La gestione degli allarmi è delegata ai servizi in ricezione delle letture

Message queue

Il blocco rosso in figura rappresenta un container Docker, il quale esegue RabbitMQ in modalità Broker topic-based di tipo Publisher\Subscriber (async).

Message handler

Il blocco in giallo è un docker container contenente i diversi message handler.

Specifiche tecniche dei message handler:

- Linguaggio: Python \ JavaScript
- Funzionamento generale: l'handler effettua il *subscribe* al topic e riceve i messaggi (in modo asincrono) dalla coda; li elabora ed invia il risultato al Data Manager tramite REST

Analyzer

Un web server espone il *front-end* agli utenti, dal quale è possibile visualizzare lo stato attuale e le statistiche dei server monitorati. In più, offre un proxy verso il data manager limitando le richieste inoltrabili dai client.
Osservando anche dal punto di vista della sicurezza, questa scelta permette di isolare il segmento di rete dell'analyzer, accessibile dall'esterno. È quindi negato agli stessi utenti di poter raggiungere il resto del sistema distribuito (isolamento App layer + Net layer). In altre parole, l'analyzer può raggiungere il Data Manager e gli user possono raggiungere solo analyzer.

Front-end

- Framework: AngularJS + Material + Angular-Chart
- Funzionamento generale: esegue il fetch dei dati dal *back-end* (tramite REST) e mostra i risultati all'utente, anche generando grafici

Data manager

Gruppo di docker container, rappresentato dal blocco viola, che contiene il Data Manager. Riceve dai message handler, comunica via REST con il backend dell'analyzer e con i replica manager.

Specifiche tecniche:

- Linguaggio: Java EE
- Si occupa della gestione dei dati su DHT
- È un client della network DHT Chord-inspired, posta a livello inferiore, e non fa parte del ring, come reso possibile dal protocollo [5]

Database e Replica Manager

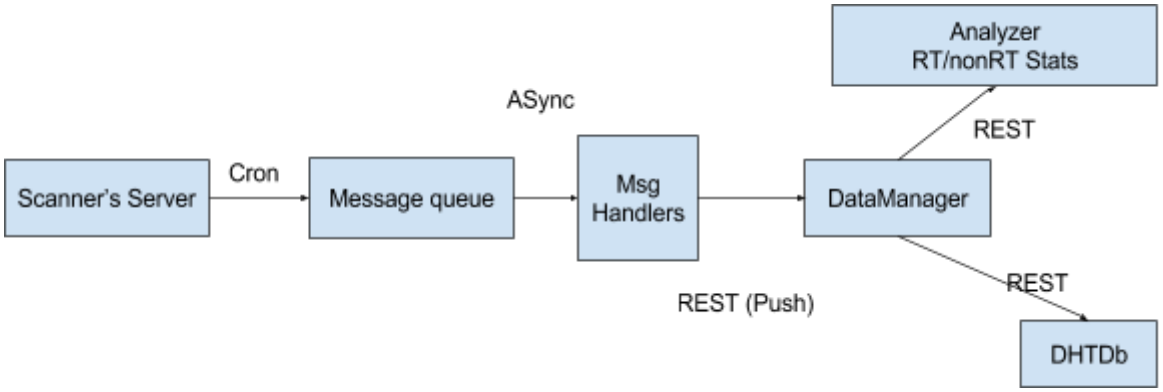
Gruppo di docker container, rappresentato dal blocco azzurro, che contiene i Replica Manager.

Il Database scelto, da distribuire su DHT, è MongoDB per le proprietà di consistenza non strong.

Specifiche tecniche:

- Linguaggio: Java EE
- DHT di tipo Chord-inspired
- Database locale: MongoDB
- REST

La gestione delle repliche è posta al livello immediatamente superiore ai database che si mantengono l'uno dall'altro indipendenti (ad es. dal punto di vista dello storage engine [4]).



DHT, Chord & indexing

Distribuzione e accesso ai dati

L'idea è quella di distribuire i dati sui diversi nodi\repliche organizzati con CHORD [5].
Le operazioni di base da implementare sono:

- Insert\Lookup: fornite dal protocollo di base Chord
- Range queries: individuiamo 2 dimensioni, un identificativo della macchina monitorata e il tempo; sarebbe utile avere a disposizione una strategia per effettuare, mantenendo uniforme la distribuzione dei dati, range query del tipo *get(timeInterval, machineId)* con almeno uno dei due parametri opzionale (come wildcard)
- Join\Leave

Le strategie proposte in letteratura per raggiungere quest'ultimo scopo si suddividono in:

- *in-DHT*: prevede la modifica del protocollo DHT sottostante, risulta efficiente ma poco scalabile [2];
- *over-DHT*: meno efficiente, più scalabile ed eredita load\data balancing dal protocollo dht sottostante.

Quest'ultima è la strada scelta nella proposta di (m)LigHT, di cui forniamo una breve sintesi (fare riferimento a [2], [6]).

LigHT

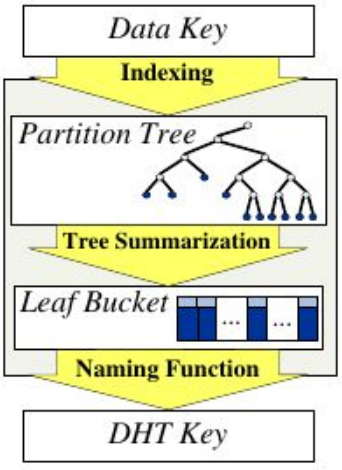


Fig. 1: LigHT Indexing Architecture

LigHT costituisce un ulteriore layer, da porre al di sopra della DHT, che svolge una funzione di indexing, basata su un index mapping ad albero binario, efficiente per query complesse.
mLigHT estende LigHT a spazi di chiavi multi dimensionali.

LigHT è, in effetti, una evoluzione di strategie over-DHT precedenti e correlate: PHT, RST e DST ([7] [8] [1]).

Definito *record* una unità di dato, questo è identificato da una *data key δ* ed ha associata, al fine di assegnare i record al DHT sottostante, di una seconda chiave, non univoca per record (ma univoca per *bucket leaf*) e chiamata *DHT Key* (κ), ottenuta per mezzo di una una *naming function* f_n .
Data una DHT Key κ, questa è associata al peer avente *id* maggiore tra quelli minori di *hash(κ)*.

Nota: nel nostro caso, il record è costituito dalla coppia (mongoDocument.time, mongoDocument), dove .time indica il tempo di registrazione del dato dallo scanner.

Consideriamo dapprima il caso unidimensionale e, successivamente, l'idea di base dell'estensione a due dimensioni.

(Space) Partition tree e naming function

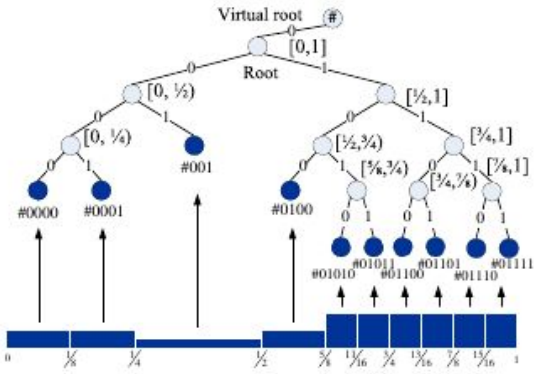


Fig. 2: An example of a space partition tree

Lo *(space) partition tree* è l'albero (virtuale) che mantiene l'indexing. Partiziona ricorsivamente lo spazio dei dati in due sottospazi di uguale dimensione. Ogni sottospazio viene ulteriormente dimezzato quando contiene un numero di data record maggiore di una threshold (θ_{split}) e compattato quando ne contiene un numero minore di (θ_{merge}).
L'albero, inoltre, possiede una *virtual root*, aggiuntiva, identificata da una propria label, qui indicata con #: la virtual root garantisce la proprieta' di "completeness" dell'albero (il numero di foglie e' uguale al numero di nodi interni), e si puo' dimostrare che la funzione di naming e' biettiva, proprieta' fondamentale per permettere di ottenere da una label l'immagine su un nodo interno e, viceversa, la controimmagine.

L'albero e' assolutamente astratto, nessun nodo ha conoscenza totale della sua struttura: ogni Bucket ha piuttosto una conoscenza locale dell'albero che deriva dalla sua label e dalla funzione di naming.

L'utilizzo di due chiavi per ogni record (δ, ad es. il tempo di registrazione, e κ, la DHT key) permette di mantenere la data locality nell'albero.

Si considera una struttura dati, *leaf bucket,* per conservare i record e sintetizzare le informazioni strutturali del partition tree.
Ogni *leaf bucket* consiste di una *leaf label* e un insieme di copertura dello spazio delle chiavi.

Data una label λ, l'opportuna DHT key è ottenuta tramite la funzione di naming, come segue:

$$\kappa = f_n(\lambda)$$

La *label* di una foglia può essere determinata, a partire dal partition tree, seguendo il percorso dalla virtual root alla foglia e concatenando i bit associati agli archi incontrati nel percorso. In particolare, è associato il bit 0 all'arco dal nodo padre al figlio di sinistra, e il bit 1 verso quello di destra.

Ad esempio in fig.2 la foglia più a destra ha label *#01111*, concatenando i bit incontrati dalla virtual root alla foglia, scendendo sempre verso destra.

La f_n proposta tronca i bit uguali consecutivi alla fine della label. Formalmente, per una dimensione:

$$f_{1d}(b_1, \ldots, b_i) = \begin{cases} f_{1d}(b1, \ldots, b_{i-1}) & \text{se } b_{i-1} = b_i \\ b_1 \ldots b_{i-1} & \text{altrimenti} \end{cases}$$

Lookup

Sia δ una data key, il lookup consiste nel calcolare λ(δ), la *label* del *leaf bucket* che copre δ. λ(δ) deve essere un prefisso della rappresentazione binaria di δ, più precisamente il più lungo prefisso che corrisponde ad una foglia esistente nel *partition tree*.
Detta *D* la massima altezza dell'albero (derivabile da una stima della dimensione e della distribuzione del data set), un possibile prefisso ha lunghezza compresa tra 2 e *D* + 1.

Il problema del lookup diviene trovare la più lunga λ(δ) tra i D prefissi candidati.
Il client può effettuare una ricerca (binaria) come sotto:

Algorithm 1 LigHT-lookup(data key δ)

```
1: μ ←binary-convert(δ)
2: lower← 2, upper←D + 1
3: while lower ≤ upper do
4:     mid←(lower+upper)/2
5:     x← μ.prefix(mid)
6:     bucket_label← DHT-get(f_n(x))
7:     if bucket_label=NULL then {a failed DHT-get}
8:         upper←f_n(x).length
9:     else if bucket_label covers δ then {reach the target leaf bucket}
10:        return f_n(x)
11:    else {x is an ancestor of the target leaf node}
12:        lower← f_nn(x,μ).length
13: return NULL
```

Ottenuta la label λ del nodo che copre δ, $f_n(\lambda)$ restituisce la DHT key κ. Il lookup del data record con data key δ va effettuato sul peer avente l'*id* maggiore tra quelli minori di *hash(κ)*.
A riga 12, viene usata una funzione simile alla naming function che permette di incrementare lower in maniera ottimale al fine di diminuire le iterazioni con le relative lookup.
Nell'implementazione si e' utilizzata una ulteriore chiave per rappresentare i dati del bucket, hash della chiave della foglia κ concatenata con un suffisso noto al datamanager: si e' cosi migliorato il flusso di traffico nella rete.

Insertion e distribuzione del carico di dati

Data δ, si esegue un lookup che restituisce il *target leaf bucket* λ(δ); dunque, in modo analogo al lookup, si accede al nodo corrispondente. In sintesi:

DHT-put(hash(f_n(λ(δ))), myRecord)

Se il leaf bucket è gia pieno, l'insertion implica uno split del bucket, operazione che genera due nuove foglie.
Lo split risulta incrementale, in quanto una delle foglie figlie, detta *local leaf*, mantiene la metà dei dati, mentre l'altra, detta *remote leaf,* riceve solo l'altra metà dei dati.

Algorithm 2 leaf-split(leaf bucket b)

1: $\lambda \leftarrow b.leaflabel$
2: **if** $\lambda = p011*$ **then**
3: $rb.leaflabel \leftarrow \lambda 0$ {rb is the remote leaf bucket}
4: $b.leaflabel \leftarrow \lambda 1$
5: **else**
6: $rb.leaflabel \leftarrow \lambda 1$
7: $b.leaflabel \leftarrow \lambda 0$
8: Add corresponding records to rb and delete them in b.
9: Locally write b back to the disk of current peer.
10: DHT-put(λ, rb)

Range query

Per eseguire una range query, è necessario trovare il *lowest common ancestor* (LCA), ovvero il nodo a profondità maggiore che copre il range desiderato, da cui deriva un *local tree* di nodi ai quali inoltrare le *DHT-get(.)*.

Il procedimento è descritto di seguito:

Algorithm 4 recursive-forward(bucket b, range R)

1: $leftwards \leftarrow (b.leaflabel = p011*)$
2: $\beta \leftarrow b.leaflabel$
3: **loop**
4: **if** $leftwards = true$ **then**
5: $\beta \leftarrow f_{ln}(\beta)$
6: **else**
7: $\beta \leftarrow f_{rn}(\beta)$
8: $inv \leftarrow$ interval(β) {compute the interval covered by branch node β}
9: **if** $inv \cap R = NULL$ **then**
10: **return**
11: **else if** $inv \subseteq R$ **then**
12: $nextbucket \leftarrow$ DHT-lookup($f_n(\beta)$)
13: recursive-forward($nextbucket$, inv)
14: **else**
15: $nextbucket \leftarrow$ DHT-lookup(β)
16: **if** $nextbucket = NULL$ **then** {a failed DHT-lookup}
17: $nextbucket \leftarrow$ DHT-lookup($f_n(\beta)$)
18: recursive-forward($nextbucket$, $inv \cap R$)
19: **return**

Algorithm 5 general-forward(range R)

1: $LCA \leftarrow$ computeLCA(R).
2: $bucket \leftarrow$ DHT-lookup($f_n(LCA)$)
3: **if** $bucket = NULL$ **then** {a failed DHT-lookup}
4: **return** LigHT-lookup($R.lowerbound$)
5: **else**
6: **if** $bucket$ overlaps R **then** {turn into the simple case}
7: **return** recursive-forward(R, $bucket$)
8: **else**
9: $bucket \leftarrow$ DHT-lookup($LCA0$)
10: $result_0 \leftarrow$ recursive-forward($R \cap bucket.range$, $bucket$)
11: $bucket \leftarrow$ DHT-lookup($LCA1$)
12: $result_1 \leftarrow$ recursive-forward($R \cap bucket.range$, $bucket$)
13: **return** $result_0 \cup result_1$

Estensione a 2 dimensioni

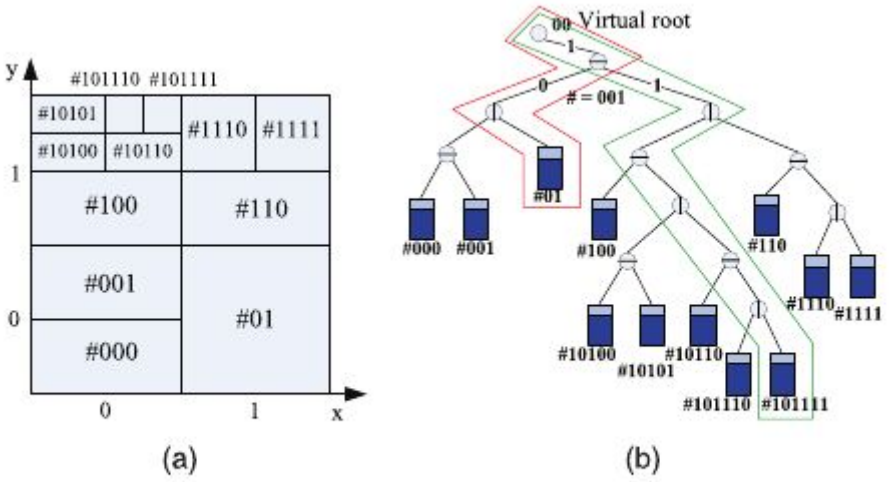


Fig. 2. Space kd-tree. (a) Space partitioning. (b) Space kd-tree decomposition (with each box corresponding to a local tree).

Lo spazio dei dati è partizionato in celle lungo le diverse dimensioni.
Similarmente alla versione unidimensionale, ogni cella contiene al più θ_{split} data records e almeno θ_{merge}.

Naming Function (2D)

La naming function f_{2n} proposta, per il caso bidimensionale, è:

$$f_{2d}(b_1 \ldots b_i) = \begin{cases} f_{2d}(b_1 \ldots b_{i-1}) & \textit{se } b_{i-2} = b_i \\ b_1 \ldots b_{i-1} & \textit{altrimenti} \end{cases}$$

Lookup

In modo simile al caso monodimensionale, data una chiave $\delta = <\delta_1, \delta_2>$, ottenuta cioè per concatenazione delle rappresentazioni binarie delle due chiavi, si cerca la *target leaf bucket* $\lambda(\delta)$, la *label* del *leaf bucket* che copre δ.

Range query

Algorithm 1. range-query(range R)
1: $\omega_R \leftarrow$ lowest-common-ancestor(R)
2: $\lambda \leftarrow$ DHT-lookup($f_{md}(\omega_R)$)
3: **if** $\lambda == NULL$ **then**
4: **return** lookup(R.top_left_corner)
5: **else if** $R \subseteq \lambda$ **then**
6: **return** the keys of λ that are in the range R
: **else**
: **return** recursive-forward(R, ω_R)

gorithm 2. recursive-forward(range R, region β)
: $\lambda \leftarrow$ DHT-lookup($f_{md}(\beta)$)
: **for all** $\beta_i \in$ {branch nodes between λ and β} **do**
: $R_i \leftarrow \beta_i \cap R$
4: **if** $R_i \neq NULL$ **then**
5: recursive-forward(R_i, β_i)

Algorithm 3. parallel-recursive-forward(range R, region β, lookahead steps h)

```
1: λ ← DHT-lookup(f_md(β))
2: for all β_i ∈ {branch nodes between λ and β} do
3:     for all β_{i,j} ∈ {leaves of subtree rooted at β_i, of depth h} do
4:         R_{i,j} ← β_{i,j} ∩ R
5:         if R_{i,j} ≠ NULL then
6:             parallel-recursive-forward(R_{i,j}, β_{i,j}, h)
```

Spazio dei dati non limitato

I leaf bucket agli estremi dell'intervallo (-inf, i1) e (i2, +inf), vengono mantenuti nella radice, per ragioni di load avoiding, si prevede di usare come threshold $\theta_{split}/2$ esclusivamente per la DHT key “#”. [6]
Non implementato.

Scalabilità

Dipende da θ_{split}. Se troppo alto, risulta inutile per pochi dati (che andrebbero su un solo peer), se troppo basso si ottiene un alto grado di distribuzione, conveniente per pochi dati.
θ_{split} determina anche il grado di uniformità della distribuzione dei dati.

Lookup e query su chord

Function	Description
`insert(key, value)`	Inserts a `key/value` binding at r distinct nodes. Under stable conditions, exactly r nodes contain the key/value binding.
`lookup(key)`	Returns the value associated with the key.
`update(key, newval)`	Inserts the `key/newval` binding at r nodes. Under stable conditions, exactly r nodes contain key/newval binding.
`join(n)`	Causes a node to add itself as a server to the Chord system that node n is part of. Returns success or failure.
`leave()`	Leave the Chord system. No return value.

Table 1: API of the Chord system.

Il lookup restituisce l’intero bucket e potrebbe bastare, al limite nei leaf bucket agli estremi ritroviamo qualche dato superfluo, in uno a sinistra, nell’altro a destra

Riferimenti

[1] Distributed Segment Tree: Support of Range query and cover query over DHT (Zheng et al.) [https://tinyurl.com/yd8declm]
[2] m-LIGHT: A LIGhtweight multidimensional index for Complex Queries over DHTs (Tang et al.) [https://tinyurl.com/y7tdgcag]
[3] https://wiki.apache.org/cassandra/ArchitectureSSTable
[4] Layering a DBMS on a DHT-Based Storage Engine (Ribas et al.) [https://tinyurl.com/ybkscqzk]
[5] Chord: A scalable peer-to-peer lookup service for Internet applications (Stoica et al.) [https://tinyurl.com/y9uf5opr]
[6] LIGHT: A query-efficient yet low-maintenance indexing scheme over DHTs (Tang et al.) [https://tinyurl.com/ydaxultx]
[7] Prefix Hash Tree: An indexing Data Structure over Distributed Hash Tables (Ramabhadra et al.) [https://tinyurl.com/ybu3e922]
[8] An Adaptive Protocol for Efficient Support of Range queries in DHT-based Systems (Gao et al.) [https://tinyurl.com/yc6l6n7s]