JISKa, Vol. 3, No. 3, Januari, 2019, Pp. 145 – 155
ISSN : 2527 – 5836 (print) | 2528 – 0074 (online)

Metode-Metode Optimasi *Memcached* sebagai NoSQL *Key-value Memory Cache*

Mandahadi Kusuma, M.Eng
Program Studi S1 Teknik Informatika
Universitas Islam Sunan Kalijaga Yogyakarta
Email : 19841115201903100 3@uin-suka.ac.id

Abstract

Memcached is an application that is used to store client query results on the web into the memory server as a temporary storage (cache). The goal is that the web remains responsive even though many access the web. Memcached uses key-value and the LRU (Least Recenly Used) algorithm to store data. In the default configuration Memcached can handle web-based applications properly, but if it is faced with an actual situation, where the process of transferring data and cache objects swells to thousands to millions of items, optimization steps are needed so that Memcached services can always be optimal, not experiencing Input / Output (I / O) overhead, and low latency. In a review of this paper, we will show some of the latest research in memcached optimization efforts. Some methods that can be used are clustering are; Memory partitioning, Graphic Processor Unit hash, User Datagram Protocol (UDP) transmission, Solid State Drive Hybird Memory and Memcached Hadoop distributed File System (HDFS)
Keywords : memcached, optimization, web-app, overhead, latency

Memcached merupakan aplikasi yang digunakan untuk menyimpan hasil *query client* pada web kedalam *memory server* sebagai penyimpanan sementara (*cache*). Tujuannya agar web tetap responsif meskipun banyak yang mengakses web tersebut. Memcached menggunakan *key-value* dan algoritma LRU (*Least Recenly Used*) untuk menyimpan data. Pada konfigurasi defaultnya *memcached* dapat menangani aplikasi berbasis web dengan baik, namun apabila sudah berhadapan dengan situasi yang sebenarnya, dimana proses transfer data dan *object cache* membengkak menjadi ribuan hingga jutaan item, perlu dilakukan langkah-langkah optimasi agar *service memcached* dapat selalu optimal, tidak mengalami *Input/Output (I/O) overhead*, dan rendah *latency*. Pada review paper ini akan menunjukkan beberapa penelitian terbaru dalam upaya optimasi *memcached*, Beberapa metode yang dapat dipakai adalah clustering, *Memory partitioning, Graphic Processor Unit hash, User Datagram Protocol (UDP) transmission, Solid State Drive Hybird Memory and Memcached Hadoop distributed File System (HDFS).*
Kata kunci : memcached, optimization, web-app, overhead, latency

1. PENDAHULUAN

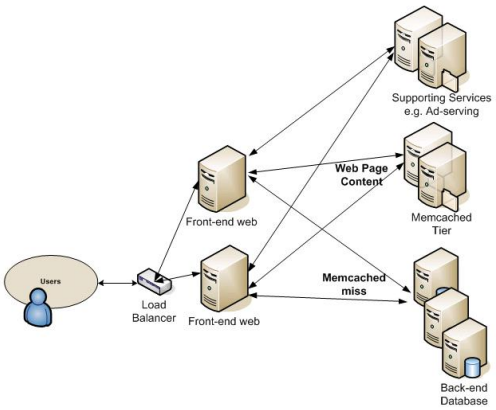
Untuk meningkatkan kecepatan akses pada aplikasi berbasis web digunakan *cache*. *Web cache* berada diantara web *server* dan *client*. Ketika ada *request* dari *client*, *cache* akan menyimpan *response*, dalam bentuk html, gambar, atau *file*. Selanjutnya apabila ada *request* yang sama dari sisi *client*, maka *cache server* akan langsung memberikan *response*, tidak perlu kembali mengakses web *server* asli.(“Caching Tutorial for Web Authors and Webmasters” n.d.)

Dari beberapa jenis teknologi *cache* yang dikembangkan, jenis *cache Memory Cache* saat ini digunakan oleh beberapa perusahaan IT terkemuka seperti Facebook, Twitter, Amazon dan Amazon. Perusahaan tersebut menggunakan *Memcached* untuk menangani jutaan *query* perdetik (“Taking In-Memory NoSQL to the Next Level” n.d.) Karena terbukti *Memory cache* dapat meningkatkan performa aplikasi web secara signifikan.

Performa *server cache* meningkat karena *response cache* tidak disimpan didalam *hardisk*, yang notabenya jauh lebih lambat dibanding CPU, melainkan disimpan langsung didalam *Memory server cache* atau yang biasa dikenal sebagai RAM. Penyimpanan yang dilakukan menggunakan NoSQL dengan tipe *key-value data store*.

Ada beberapa aplikasi *memory cache* yang populer, salah satunya adalah *memcached*. *Memcached* populer digunakan karena memiliki fitur mudah untuk *clustering*. Sehingga apabila

proses *cache* bisa dibagi, dan apabila salah satu *server memcached* bermasalah/*down* tidak akan menghentikan layanan secara keseluruhan. Gambar 1 menunjukkan arsitektur *memcached* secara umum dan alur pengambilan data dari *client*



Gambar 1. arsitektur Memory cache

Namun untuk data besar dengan pengakses ribuan hingga jutaan *client*, *default* konfigurasi pada *memcached* sering menimbulkan masalah, seperti I/O *overhead*, *memcached server* melambat dan *miss hit cache* (“Taking In-Memory NoSQL to the Next Level” n.d.). Berbagai metode yang telah diujicoba dan diimplementasikan oleh peneliti pada *memcached* bertujuan untuk menurunkan *latency* dan *overhead* serta meningkatkan performa aplikasi *memcached*. Dalam paper ini akan dibahas lebih lanjut mengenai perngertian NoSQL *Database* , Metode penyimpanan, *Key-value*, dan mekanisme *memcached.*

1.1 NOSQL (NOT ONLY SQL) DATABASE

Secara umum *Database* NoSQL adalah alternatif dari *database* relational yang umum digunakan, dengan kemampuan utama *scalability*, *avaibility* dan *fault tolerance*. Kemampuan NoSQL jauh diatas relasional *Database* dan sangat sesuai dengan kebutuhan aplikasi bisnis saat ini. (“NoSQL Databases Defined & Explained | Planet Cassandra” n.d.). NoSQL lahir karena para *developer* aplikasi dipusingkan dengan masalah ketidaksesuaian struktur data didalam *Database* relational dengan struktur data didalam aplikasi. Menggunakan NoSQL *developer* tidak perlu lagi melakukan penyesuaian antara struktur data aplikasi dengan struktur data pada *Database* (“NoSQL Databases: An Overview | ThoughtWorks” n.d.).

Dari sudut pandang bisnis, lingkungan NoSQL atau bisa juga disebut sebagai lingkungan ‘*Big Data*’ telah memperlihatkan keunggulan yang sangat kompetitif dan memiliki keuntungan dalam berbagai bidang di industri. Ada 4 tipe NoSQL *Database*, *Key Value store*, *column store*, document *Database*, dan graph *Database*. Selanjutnya pada pembahasan akan berfokus pada *Key-value store* yang dimanfaatkan sebagai *Memory Cache* pada aplikasi berbasis web.

1.2 KEY-VALUE STORE

Key Value Store adalah sejumlah *hashmap* data terstruktur yang disimpan dan dapat diakses nilainya menggunakan *key*. Setiap *record* memiliki panjang dan nilai data yang berbeda-beda. Contoh *key-value* tidak terstruktur ditunjukkan pada gambar 2.

Setiap *record* memiliki *key*, dan menyimpan jenis *field* apa saja disebut sebagai *bins* (biasa disebut sebagai *column* pada *database* terstruktur). Setiap *bins* terdiri dari nama *field* dan *value*. Untuk setiap informasi data yang dmiliki dapat dibuat *bin*. Apabila tidak terdapat informasi data tertentu, *field* kosong tidak perlu dibuat. Setiap *record* dimungkinkan menyimpan jenis dan *value* data yang berbeda-beda tergantung informasi yang dimiliki setiap *record* tersebut.

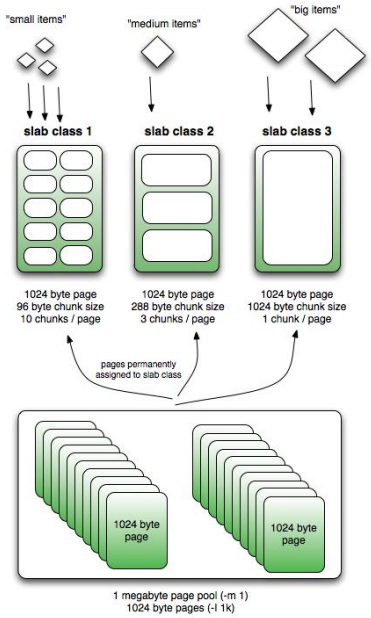
Disebut *key-value store* karena setiap *record* memiliki primary *key* dan kumpulan *value* (*bins*). Dapat juga disebut sebagai *row Store* karena semua data untuk sebuah *record* disimpan bersama-sama. (“What Are NoSQL Key-Value Store Databases?” n.d.)

Key: 1	ID: sj	First Name: Sam	
Key: 2	Email: jb@gmail.com	Location: London	Age: 37
Key: 3	Facebook ID: jkirk	Password: xxx	Name: James

Gambar 2. Contoh Key-value (data tidak terstruktur) pada record informasi user

1.3 PENGENALAN MEMCACHED

Awal mulanya dikembangkan oleh Brad Fitzpatrick untuk livejournal pada tahun 2003. *Memcached* adalah *high-performance system memory cache object* yang terdistribusi bertujuan untuk meningkatkan kecepatan akses aplikasi web dengan mengurangi *load* beban *database* utama. Bisa dibayangkan sebagai pengingat jangka pendek terhadap web aplikasi yang dimiliki. (“Memcached - a Distributed Memory Object Caching System” n.d.)

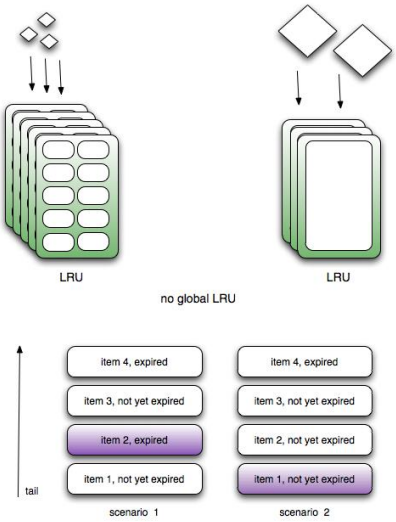


Gambar 3. ilustrasi alokasi slab (block) didalam Memory

Seperti yang ditunjukkan pada gambar 3, *Memcached* menggunakan algoritma LRU (*Least Frequency Used*) pada setiap *slab*(*block*) *memory* dalam proses penyimpanan data sementara. Secara umum algoritma LRU memastikan *object* yang dibuang adalah *object* yang sudah tidak aktif atau yang paling lama yang pernah digunakan, kemungkinan besar data *cache* sudah tidak update (out of date). Jadi apabila *server* kehabisan *memory*, *slabs memory* yang *expired* akan dibuang terlebih dahulu, selanjutnya yang dibuang adalah *slabs* yang terlama. Apabila sistem tidak memiliki data yang *expired*, maka yang akan dibuang adalah *slabs* (*block*) paling lama tidak dipakai (“Memcached LRU and Expiry - Stack Overflow” n.d.).

Sperti yang terlihat pada gambar 4, karena setiap *slab class* sebenarnya adalah *cache* yang terpisah, maka tidak ada global LRU, yang ada hanya LRU pada masing-masing *slab class*. *Object* yang paling lama tidak digunakan tidak akan dibuang apabila tidak berada pada *slab class* yang sama. Tergantung pada pola akses penyimpanan yang digunakan, bisa saja ada satu atau lebih *slabs* yang selalu membuang *item* yang masih digunakan, sementara pada *class*

slab lain masih tersimpan *item-item* yang sudah *expired* (“{ Work }: Memcached for Dummies” n.d.)

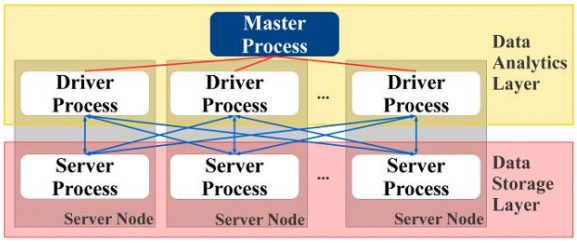


gambar 4. algoritma LRU pada memcached

1.4 ANALISIS PERFORMA MEMCACHED

Performance analisis yang dilakukan baik pada *Performance operation* dan *object operation* seperti *set/GET* pada konfigurasi *memcached default* tidak dapat ditangani dengan efisien. CPU dan performa I/O pada tcp *stack* mengalami *bottleneck*, baik pada *single server* maupun menggunakan arsitektur *server* cluster. (Zhang et al. 2014)

Test performa menggunakan arsitektur *memcached* seperti pada gambar 5, dilakukan menggunakan algoritma *pagerank* yang diimplementasikan didalam *map*/*reduce style*. Pada Fase *map* dihitung *rank* yang telah dikontribusikan untuk halaman lain pada setiap halaman web. Pada fase *reduce*, setiap *node* dihitung *rank* baru untuk setiap halaman web lokal berdasarkan *rank* yang telah dikontribusikan. Oleh karena itu diperlukan mekanisme yang baik agar management data didalam *memory* dapat lebih baik. (Zhang et al. 2014).



Gambar 5. Performance architecture pada memcached

2. METODE PENULISAN

Penulisan ini menggunakan studi kepustakaan yang penulis ambil meliputi web dan artikel penelitian ilmiah tentang optimasi sistem *memcached*. Disertai sumber media lain berupa internet yang berhubungan dengan pengertian umum tentang *memory cache*.

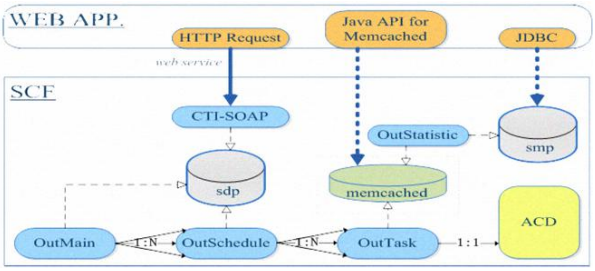
3. PEMBAHASAN

Berdasarkan uraian diatas berikut akan dijabarkan metode-metode optimasi yang telah digunakan pada *memory cache*d

3.1 WEBSERVICE DAN MEMCACHED

Mem*cache* tidak harus selalu menggunakan web *server*. Karena sifatnya yang terbuka, maka mem*cache* juga memiliki API ke program lain, seperti JAVA melalui *webservice*. Sebagai contoh pada penelitian yang memanfaatkan mem*cache* untuk meningkatkan proses *read* pada aplikasi *call center* yang digunakan pada perusahaannya. Seperti pada gambar 6, menunjukkan desain *webservice* menggunakan *memcached.*

Karena *Database* selalu mengalami *load* yang tinggi pada saat dilakukan *outbound call*, maka diujicoba mekanisme mem*cache* untuk proses caching informasi yang diperoleh didalam *memory*.



Gambar 6. memcached dan Webservice

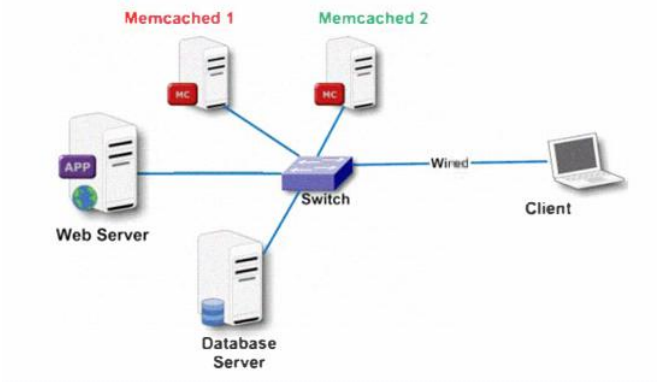
Peneliti mengakses mem*cache* menggunakan aplikasi berbasis java untuk mengakses data *client* diwebsite. Karena *Memcached* pada dasarnya merupakan *webservice*, hal ini memudahkan beberapa aplikasi yang menggunakan bahasa pemograman yang berbeda dapat saling berkomunikasi seperti pada gambar 6. Selain itu juga Mem*cache* bekerja secara *transparent*, sehingga lebih aman dan fleksibel. Berdasarkan percobaan yang dilakukan oleh peneliti, meskipun menggunakan *webservice*, tetap dapat meningkatkan performa dari *Call center* secara signifikan. (Li, Zhan, and Li 2011).

3.2 CLUSTERING MEMCACHED

Memcached memiliki kemampuan untuk dilakukan cluster. Tujuannya membagi beban proses pada aplikasi *memcached* tersebut dan sebagai *failover* apabila salah satu *server* mengalami kerusakan/*downtime*.

Bakar, dkk melakukan evaluasi performa terhadap sistem cluster pada *memcached*. Tujuannya untuk mengetahui kemampuan cluster *memcached* dalam meningkatkan performa kecepatan akses web secara kuantitatif. Riset yang dilakukan juga membandingkan performa antara *server memcached* yang berdiri sendiri (*single*) dengan *clustered memcached*.

Arsitektur *memcached* yang digunakan dalam percobaan tersebut terdiri dari 2 server *memcached* yang bekerja secara *redundant* seperti pada gambar 7. Sedangkan spesifikasi sistem dan server yang digunakan ditunjukkan pada tabel 1 dan 2



Gambar 7. arsitektur clustered memcached

Apache Web Server
Ubuntu Server 64-bit version 9.10 PHP Version 5.2.10-2ubuntu6.4 Apache 2.2.12 memcached extension 2.2.5
MySQL Server
Ubuntu Server 64-bit version 9.10 mysql-server 5.1
Memcached Server(s)
Ubuntu Server 64-bit version 9.10 memcached version 1.4.4 libevent 1.4-2
Client
Ubuntu Desktop version 9.10 Apache Bench 2.3
Database Specification
Employee Sample database, developed by Patrick Crews and Giuseppe Maxia for testing Six separate tables 4 million records in total (approximately 160MB)

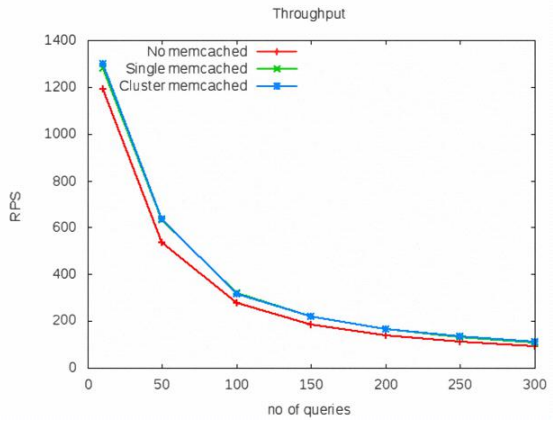
TABLE 1 SPESIFIKASI SISTEM

Server Specification	
Processor	AMD Phenom X4 9600B, 2.33 GHz
RAM	1.7GB, DDR2
NIC	Ethernet Gigabit
Hard Disk	SATA Hard Drive
Client Specification	
Model	Acer Aspire 5570 Laptop
Processor	Intel Dual Core T2300, 1.66 GHz
RAM	2.5GB, DDR2
NIC	Ethernet Gigabit
Hard Disk	SATA Hard Drive
Network Specification	
Switch	D-Link DES-1008D 8-Port 10/100 Ethernet

TABLE 2 SPESIFIKASI HARDWARE

Test yang dilakukan adalah membuat seribu *request* dari *client* ke web *server*. Sementara *memcached* dikonfigurasi untuk dapat menampung seratus *object*s. *Request* dari *client* disimulasikan menggunakan *apachebench*. Selanjutnya *apachebench* akan mengukur *Request per Second* (RPS) *http data throughput*. Data RPS menunjukkan web *server* dapat menangani jumlah *request* perdetik. Selain itu juga dilihat penggunaan CPU dan *memory server* selama proses *benchmark* berlangsung

Dari hasil yang didapat pada gambar 8, ternyata nilai RPS baik antara *single* dan *clustered memcached* tidak menunjukkan perbedaan yang signifikan. (Bakar, Shaharill, and Ahmed 2010)



Gambar 8. Grafik hasil benchmark

3.3 MEMORY PARTITIONING

Cara kerja *memcached* adalah dengan membagi *Memory* kedalam beberapa *class* yang berbeda secara proses sesuai *request* ukuran *object* yang berbeda-beda. Ketika semua memori telah dialokasikan, tidak dimungkinkan ada proses pengalokasian kembali atau kemungkinan tersebut sangat terbatas. Masalah ini disebut sebagai *calcification*. *Calcification* dimungkinkan tersebut apabila ada perubahan pada data *object* yang diminta.

Satuan dasar *Memory* pada *Memcached* disebut sebagai *Slab* dan memiliki ukuran yang sama untuk menyimpan *object*. Didalam *Slab* dibagi-bagi menjadi *chunks*, untuk menyimpan data *items*/*object*s. Ukuran *chunks* dan jumlah *chunks* tergantung pada *class* yang digunakan oleh *slab*.

Ketika semua seluruh *slab* pada *Memory* telah digunakan oleh *class*, maka *Memcached* menggunakan aturan *Least Recently Used* (LRU) untuk proses menghapus. LRU bekerja berdasarkan *class*, sehingga *class* lain yang tersimpan didalam *chunks* dengan ukuran yang berbeda tidak akan terpengaruh/dipindahkan.

Pada saat beberapa bagian dari *Memory* telah digunakan oleh *class*, maka bagian tersebut akan selalu menjadi bagian dari *class* tertentu (kecuali apabila *server* restart), hal ini tidak akan masalah apabila

Ada 2 solusi yang saat ini digunakan yaitu *memcache automove* dan *twitter policies*.

Memcache automove bekerja dengan cara memeriksa setiap 10 detik *class* yang paling sering dikosongkan secara berturut-turut. Apabila ada *class* yang dikosongkan tiga kali (dalam 30 detik) berturut-turut maka *class* tersebut akan mendapatkan *slab* baru diambil dari *class* yang tidak mengalami pengosongan pada tiga kali pemeriksaan. Namun pada kenyataannya design ini tidak terlalu berguna, karena sangat kecil kemungkinan *slab* dipindahkan, karena sangat jarang ada *slab* yang tidak mengalami pengosongan dalam waktu 30 detik

Twitter policies menggunakan solusi random untuk mekanisme diatas. Apabila *slab* pada *memory* penuh, pada *memcached* akan memiliki secara acak *slab* tertentu dan segera mengosongkannya untuk dapat diisi dengan *object* yang baru.

Peneliti membuktikan bahwa 2 cara diatas terlalu kasar, sehingga diperlukan cara baru yang lebih baik. Peneliti menunjukan bahwa *calcification* memberikan dampak negatif dalam performa *hit rate*. Dari hasil eksperiment peneliti, bahwa mekanisme *calcification* yang digunakan oleh *memcache* menimbulkan masalah yaitu performa *hit rate* yang jelek.

Peneliti membuat skema baru yang disebut sebagai PSA (*Periodic Slab Allocation*). Dari hasil uji PSA tersebut ada peningkatan yang significant pada beberapa fase di *memcached*.

Algorithm 1 Periodic Slab Allocation (PSA)

1. **Input:** $\mathbf{s}$ // vector of slabs allocated to each class
2. **Input:** $\mathbf{r}$ // vector of requests in each class
3. **Input:** $\mathbf{m}$ // vector of misses in each class
4.
5. **Every** M misses **do**
6. $id_{\text{take}} \leftarrow i : (r_i/s_i) < (r_j/s_j), \forall\ r_j, s_j \in \mathbf{r}, \mathbf{s}$;
7. $id_{\text{give}} \leftarrow i : m_i > m_j, \forall\ m_j \in \mathbf{m}$;
8. MoveOneSlab$(id_{\text{take}}, id_{\text{give}})$;

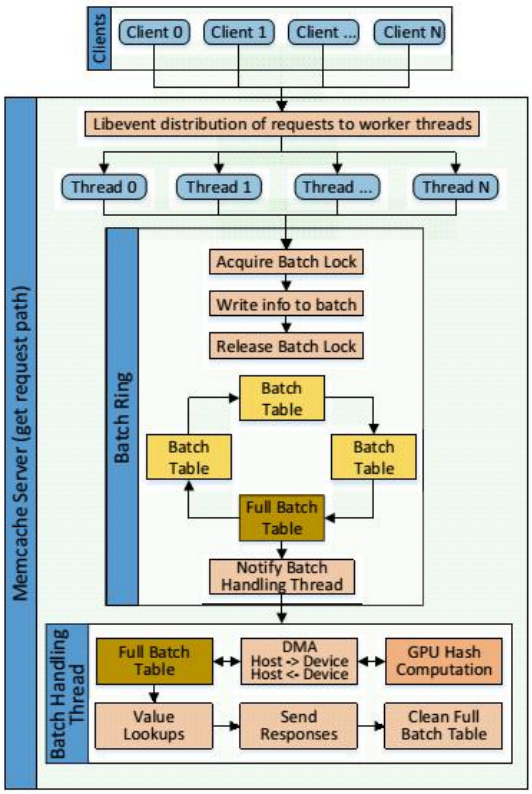
Gambar 9. Alogritma PSA

Algoritma PSA yang diimplementasikan pada *memcached* seperti pada gambar 9, ketika tidak ada *calcification*, menunjukkan ada peningkatan *hit rate* sebesar 7% , sedangkan pada saat terjadi *calcification* ada peningkatan *hit rate* sebesar 10%. Penggunaan PSA dapat meningkatkan *hit rate*, jika dibandingkan dengan mekanisme TMC (Twitter *Memcached*) untuk menangani *calcification* tapi berdampak pada *hit rate* yang rendah(Carra and Michiardi 2014).

3.4 PEMROSESAN HASHING MENGGUNAKAN GPU

Memindahkan proses hashing ke GPU, proses komputasi hashing pada mem*cache* secara *default* menggunakan *resource* CPU. Peneliti mencoba menggunakan GPU untuk proses komputasi hashing perintah *GET* yang dilakukan oleh *memcached*.

Modifikasi alur proses eksekusi yang dilakukan bertujuan untuk memanfaatkan kemampuan parallel computing yang dimiliki oleh GPU.



Gambar 10. Alurkerja Memcache GPU-assisted arsitektur untuk menangani proses GET

Seperti yang terlihat pada gambar 10, setiap proses permintaan *GET* akan ditangani oleh masing-masing th*read* terpisah yang bekerja paralel. Karena bekerja paralel, maka hasil yang didapat bisa lebih cepat. Untuk meningkatkan kemampuan paralel tersebut, digunakan pula skema double *buffer*ing. Ketika *buffer* pertama penuh, maka *buffer* akan dikirim ke *Memory* GPU yang kemudian dapat dibaca menggunakan kernel *invocation*, ketika GPU melakukan komputasi pada *key* di *buffer* yang pertama, CPU akan menduplikat *key* yang baru pada *buffer* yang kedua.

Dari hasil *benchmark* yang dilakukan penggunaan GPU pada proses hashing *key memcached* dapat meningkatkan proses 1.47-5.71 kali lebih cepat daripada *memcached* biasa (Deyannis et al. 2014).

3.5 MENINGKATKAN KECEPATAN I/O NETWORK

Dibandingkan TCP, protokol UDP lebih ringan karena jaminan pengiriman paketnya keberhasilannya lebih rendah dibanding TCP. Biasanya UDP digunakan untuk operasi yang tidak mengharuskan pengiriman data 100% berhasil. Seperti pada *request GET*, ketika pada *response*nya ada sebagian data yang hilang atau tidak lengkap maka akan dianggap *cache miss*.

Meskipun paket yang diproses kecil-kecil, tetap membutuhkan lebih banyak CPU *resource* pada networking *stack* didalam sistem operasi. Berpindah dari koneksi TCP menggunakan UDP, dapat menghasil keunggulan pada *throughput* karena TCP sendiri adalah protokol *transaction-based* selalu akan terjadi *overhead*.

Setiap proses pergerakan data dapat menyebabkan CPU *overhead*, untuk megurangi *overhead* tersebut, *user-space* UDP *stack* dapat digunakan, yaitu mem-bypass kernel sistem. *Netmap* module digunakan untuk mengirimkan paket (termasuk UDP/IP *headers*) langsung ke NIC, sehingga pertukaran paket data antar network adapter akan lebih cepat.

Namun proses *SET* dan *UPDATE* membutuhkan reliabilitas yang tinggi, apabila menggunakan UDP maka harus mengganti protokol pada mem*cache* atau menambahkan logika baru.

Hasil *benchmark* yang dilakukan proses *GET* pada *memcached* menggunakan protokol UDP, menunjukkan peningkatan yang sangat signifikan dibanding menggunakan protokol TCP (Deyannis et al. 2014).

3.6 Mem-HDFS

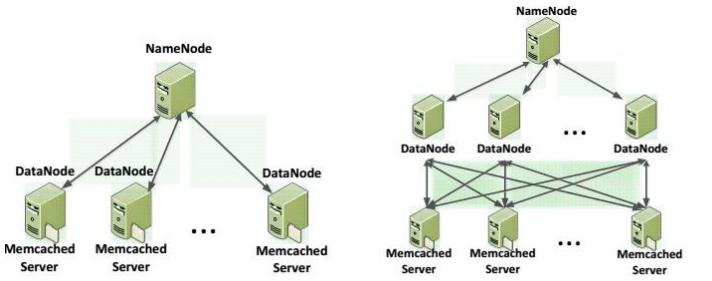
Hadoop distributed file system (HDFS) mengalami I/O *bottlenecks* dalam proses penyimpanan replikasi *datablocks*. I/O *overhead*s pada HDPS menyebabkan penurunan performa. Seperti yang terlihat pada table 3, memperlihatkan penyimpanan data dan besar bandwith yang digunakan per detik menggunakan infrastruktur *hardware* terbaru saat ini. Dari table 3 bisa disimpulkan bahwa bandwith akses data pada *memory* jauh lebih besar dibanding *bandwith* data pada *storage hardisk*. Karena secara *default* HDFS menggunakan *hardisk* fisik, jelas terdapat I/O *bottleneck* pada *default* HDFS.

	Media	Bandwidth (GB/s)
Data Storage	Hard Disk Memory (e.g. DDR3)	0.12 12.8
Data Movement	Ethernet (e.g. 40 GigE) InfiniBand (e.g. Dual-port FDR)	Up to 5 Up to 12.5

TABLE 3 DATA PADA HDFS

Penggabungan *Memcached* dengan sistem HDFS seperti yang diperlihatkan pada gambar 11 adalah salah satu upaya yang dilakukan untuk mengurangi I/O *overhead*s pada HDFS. Ketika proses HDFS *write*, *DFSClient* mengirimkan setiap *block* ke *DataNode*, kemudian *DataNode* membuat *BlockReceive*r untuk menyimpan paket data ke *memcached*. Pada saat *DFSClient* mengirimkan *request Read*s maka data dicari didalam *memcached* terlebih dahulu, apabila tidak ada baru dicari didalam HDFS.

Dari hasil proses eksperiment dan *benchmark*, dapat disimpulkan data penggunaan MEM-HDFS bisa meningkatkan performa *server* Hadoop. Dari hasil test DFSIO *write* dan *read* ada peningkatan performa sebesar 3.3x dan 3.9x dibandingkan dengan sistem Hadoop *default* tanpa menggunakan *memcached*. MEM-HDFS menjamin penggunaan *resource server* lebih efisien, selain itu juga design MEM-HDFS dapat mendukung bermacam-macam *high-Performance interconnects* yang digunakan pada cluster modern.(Islam et al. 2014)



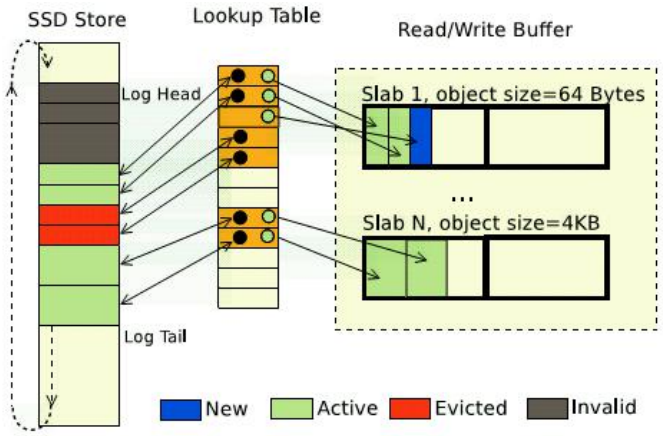
(a) MEM-HDFS-Local (b) MEM-HDFS-Remote with dedicated Memcached cluster

Gambar 11. Arsitektur Memcached dan HDFS

3.7 SSD-ASSISTED HYBIRD MEMORY UNTUK MEMCACHED

Tujuan penggunaan SSD adalah untuk menambah kapasitas *memory system*. Sehingga jumlah *memory* untuk menampung *cache* menjadi lebih besar.

Gambar 12 memperlihatkan mekanisme *hybird memory* yang terdiri dari 3 komponen utama, yaitu; 1) *Read*/*Write Buffer* untuk *cache object* yang baru saja diakses juga termasuk berbagai macam ukuran *object*. 2) *In-Memory Lookup Table* yang menyimpan semua indeks *key-value object* didalam *hybird memory*. 3) *Log Structured* SSD *storage*, adalah tempat penyimpanan seluruh *object* berada.



Gambar 12. Mekanisme hybird Memory

Penggunaan SSD dan DRAM sebagai *hybird memory* pada memungkinkan komputasi didalam menory untuk diaplikasikan kedalam sistem analytic *Big Data*. *Hybird memory* berfungsi untuk menjaga semua *file* yang diproses didalam SSD. Ketika *file* dibutuhkan, maka data blok *file* tersebut akan diambil oleh DRAM, tergantung pada access history. Hasilnya, aplikasi akan menganggap menggunakan SSD sebagai bagian dari DRAM, yaitu *memory* utama, bukan sebagai *secondary cache*. Hasil dari eksperimen menunjukkan bahwa *hybird memory* menggunakan SSD dapat ditingkatkan hingga paling tidak 2 kali besarnya DRAM yang dimiliki.(Chen et al. 2014)

Karena *hybird memory* dirancang sebagai *object allocator* yang efisien maka penggunaan *memcached* sangat cocok dilingkungan ini. Dengan sedikit penyesuaian definisi *item memcached* untuk mengakomodasi *index entry* yang dibutuhkan, maka *memcached* dapat digunakan pada *hybird memory*

Memcached menggunakan *hybird memory* dapat melampaui performa ketika SSD digunakan sebagai *swap virtual Memory*. *Read* dan *write* secara acak pada 1KB *object* 3.1x dan 3.5x lebih cepat ketika menggunakan *hybird Memory*. Ketika menggunakan *memcached*, kecepatan meningkat 3.7x dan 3.0x pada operasi *GET* dan Set. Dan t*hroughput memcached* meningkat hingga 5.3X(Ouyang et al. 2012)

4. KESIMPULAN

Kontribusi utama penelitian ini adalah menunjukkan berbagai macam percobaan dan eksperiment yang dilakukan pada *memcached* dalam upaya untuk meningkatkan performanya. Tidak semua hasil dapat digunakan bersama-sama pada satu kasus, kita harus menyesuaikan kondisi lingkungan, aplikasi yang digunakan, dan target pengguna untuk menentukan jenis optimasi yang mana yang dapat dieksplorasi lebih lanjut.

Dapat diketahui bahwa penggunaan *clustered memcached* dibandingkan dengan *single memcached* tidak memiliki perbedaan yang signifikan. Namun performa kecepatan pada akses pada memcached dapat ditingkatkan hingga 3-5 kali lipat dengan menggunakan metode *Ssd-Assisted Hybird Memory,* meningkatkan kecepatan I/O Network, atau menggunakan *resource* pada GPU.

DAFTAR PUSTAKA

"Memcached for Dummies." n.d. Accessed April 9, 2015. http://work.tinou.com/2011/04/memcached-for-dummies.html

An, Huiyao, Dunwei Liu, Lei Li, Ning Qi, Tao Yu, Peng Zhang, and Xing Zhang. 2014. "A Customized Memcached FPGA Chip for Big Data." In *2014 12th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, 1–3. https://doi.org/10.1109/ICSICT.2014.7021160

Bakar, K.A., M.H.M. Shaharill, and M. Ahmed. 2010. "Performance Evaluation of a Clustered Memcache." In *2010 International Conference on Information and Communication Technology for the Muslim World (ICT4M)*, E54–60. https://doi.org/10.1109/ICT4M.2010.5971915

"Caching Tutorial for Web Authors and Webmasters." n.d. Accessed April 10, 2015. https://www.mnot.net/cache_docs/#DEFINITION

Carra, D., and P. Michiardi. 2014. "Memory Partitioning in Memcached: An Experimental Performance Analysis." In *2014 IEEE International Conference on Communications (ICC)*, 1154–59. https://doi.org/10.1109/ICC.2014.6883477

Chen, Zhiguang, Yutong Lu, Nong Xiao, and Fang Liu. 2014. "A Hybrid Memory Built by SSD and DRAM to Support In-Memory Big Data Analytics." *Knowledge and Information Systems* 41 (2): 335–54. https://doi.org/10.1007/s10115-013-0727-6

Deyannis, Dimitris, Lazaros Koromilas, Giorgos Vasiliadis, Elias Athanasopoulos, and Sotiris Ioannidis. 2014. "Flying Memcache: Lessons Learned from Different Acceleration Strategies." In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 25–32. https://doi.org/10.1109/SBAC-PAD.2014.17.

Fukuda, E.S., H. Inoue, T. Takenaka, Dahoo Kim, T. Sadahisa, T. Asai, and M. Motomura. 2014. "Caching Memcached at Reconfigurable Network Interface." In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 1–6. https://doi.org/10.1109/FPL.2014.6927487

Islam, N.S., Xiaoyi Lu, M. Wasi-ur-Rahman, R. Rajachandrasekar, and D.K.D.K. Panda. 2014. "In-Memory I/O and Replication for HDFS with Memcached: Early Experiences." In *2014 IEEE International Conference on Big Data (Big Data)*, 213–18. https://doi.org/10.1109/BigData.2014.7004235

Li, Fajie, Shubo Zhan, and Lili Li. 2011. "Research on Using Memcached in Call Center." In *2011 International Conference on Computer Science and Network Technology (ICCSNT)*, 3:1721–23. https://doi.org/10.1109/ICCSNT.2011.6182300

"Memcached - a Distributed Memory Object Caching System." n.d. Accessed April 9, 2015. http://memcached.org/about

"Memcached LRU and Expiry - Stack Overflow." n.d. Accessed April 9, 2015. http://stackoverflow.com/questions/4962290/memcached-lru-and-expiry

"NoSQL Databases: An Overview | ThoughtWorks." n.d. Accessed April 10, 2015. http://www.thoughtworks.com/insights/blog/nosql-databases-overview

"NoSQL Databases Defined & Explained | Planet Cassandra." n.d. Accessed April 10, 2015. http://planetcassandra.org/what-is-nosql/

Ouyang, Xiangyong, N.S. Islam, R. Rajachandrasekar, J. Jose, Miao Luo, Hao Wang, and D.K. Panda. 2012. "SSD-Assisted Hybrid Memory to Accelerate Memcached over High Performance Networks." In *2012 41st International Conference on Parallel Processing (ICPP)*, 470–79. https://doi.org/10.1109/ICPP.2012.54

"Taking In-Memory NoSQL to the Next Level." n.d. Accessed April 10, 2015. https://redislabs.com/blog/taking-in-memory-nosql-to-the-next-level#.VSX79eGzbLX.

"What Are NoSQL Key-Value Store Databases?" n.d. Accessed April 10, 2015. http://www.aerospike.com/what-is-a-nosql-key-value-store/

Zhang, H., B.M. Tudor, G. Chen, and B.C. Ooi. 2014. "Efficient Inmemory Data Management: An Analysis." *Proceedings of the VLDB Endowment* 7 (10): 833–36.