Memahami Distributed Caching: Rahasia Dibalik Aplikasi Cepat

chmdznr

Daftar Isi

Daftar Gambar

1. Dasar-dasar Caching

1.1 Konsep Umum Caching

- **Definisi**: Caching adalah teknik penyimpanan data sementara di lokasi yang mudah diakses untuk mengurangi waktu respons

- **Analog**:

  - Meja kerja vs. gudang arsip

  - Gerai cepat di minimarket

- **Terminologi Penting**:

  - **Cache Hit**: Data ditemukan di cache

  - **Cache Miss**: Data tidak ditemukan di cache

  - **TTL (Time-to-Live)**: Waktu kedaluwarsa data cache

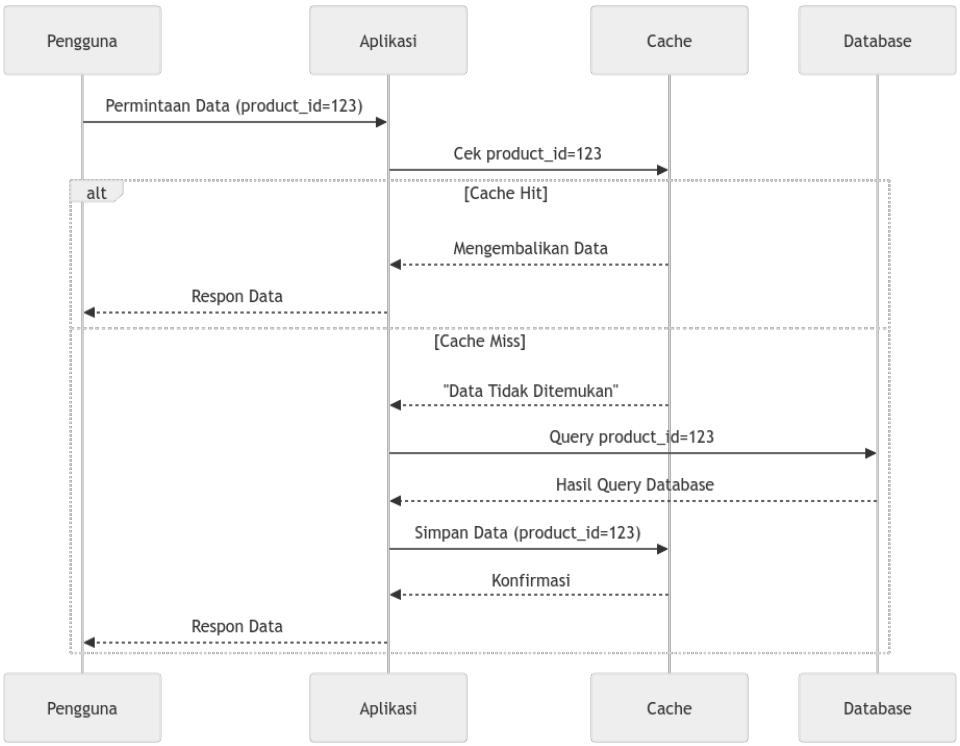
  - **Invalidasi**: Proses penghapusan data cache sebelum TTL

1.2 Perbandingan dengan Query Database Tradisional

Tabel 1.1: Perbandingan Karakteristik Caching vs Database

Parameter	**Caching**	**Database**
Latensi	Milidetik (ms)	10-100x lebih tinggi
Throughput	Ribuan operasi/detik	Terbatas I/O disk
Pola Akses	Data panas (hot data)	Data lengkap
Biaya	Memory-intensive	Storage-intensive

1.2.1 Alur Kerja Dasar



Gambar 1.1: Alur kerja sistem dengan caching

1. Aplikasi menerima permintaan data
2. Cek cache pertama:
   - Jika **hit**: Langsung kembalikan data
   - Jika **miss**: Ambil dari database
3. Simpan hasil query database ke cache
4. Kembalikan data ke pengguna

1.3 Manfaat dan Tantangan

1.3.1 Manfaat Utama

- Mengurangi latensi hingga 90%
- Menurunkan beban database
- Meningkatkan throughput sistem
- Menghemat biaya infrastruktur

1.3.2 Tantangan Umum

- **Data Stale**: Risiko data tidak update
- **Cache Stampede**: Lonjakan permintaan saat cache expired
- **Memory Management**: Alokasi memori yang efisien
- **Complexity**: Penambahan layer infrastruktur

Best Practice Awal

```
import cachetools

# Inisialisasi cache dengan TTL 5 menit
cache = cachetools.TTLCache(maxsize=1000, ttl=300)

def get_product(product_id):
        # Cek cache terlebih dahulu
        product = cache.get(product_id)
        if product:
                return product

        # Jika tidak ada di cache, query database
        product = db.query_product(product_id)

        # Simpan ke cache
        cache[product_id] = product
        return product
```

Listing 1: Contoh Implementasi Sederhana Caching

2. Konsep Distributed Caching

2.1 Apa Itu Distributed Caching?

- **Definisi**: Distributed caching adalah teknik penyimpanan data sementara yang terdistribusi di beberapa node/server, bekerja secara terkoordinasi untuk menyediakan akses cepat ke data dengan skalabilitas tinggi dan toleransi kegagalan.
- **Analog**: Bayangkan jaringan ATM yang terhubung secara global. Saldo Anda di-cache di beberapa lokasi ATM agar bisa diakses cepat di mana pun, tanpa harus selalu menghubungi server pusat.
- **Karakteristik Kunci**:
  - **Terdesentralisasi**: Data disebar di banyak node.
  - **Replikasi**: Data diduplikasi untuk redundansi.
  - **Konsistensi**: Mekanisme untuk memastikan data di semua node tetap sinkron.

2.2 Kapan Distributed Caching Dibutuhkan?

- **Skenario 1: Aplikasi Global dengan Pengguna Terdistribusi** Contoh: Aplikasi streaming seperti Netflix yang perlu menyediakan konten dari server terdekat dengan pengguna (misalnya: cache di edge server Singapura untuk pengguna Asia Tenggara).
- **Skenario 2: High Traffic Instan** Contoh: Flash sale e-commerce seperti Tokopedia atau Shopee, di mana jutaan request terjadi dalam hitungan detik. Distributed cache membantu menghindari kelebihan beban database.
- **Skenario 3: Sistem Microservices** Contoh: Layanan checkout di e-commerce yang memerlukan akses cepat ke data keranjang belanja yang tersebar di banyak service.

2.3 Keunggulan vs. Tantangan

Tabel 2.1: Perbandingan Keunggulan dan Tantangan Distributed Caching

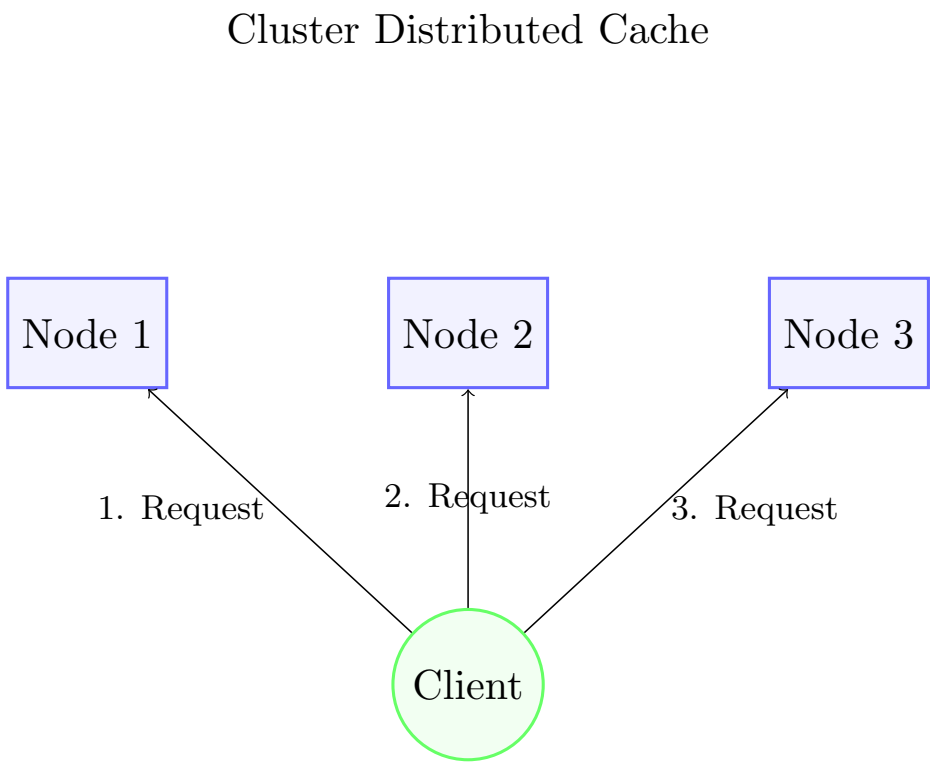
Keunggulan	**Tantangan**
Skalabilitas horizontal dengan mudah	Kompleksitas manajemen node
Reduksi latensi melalui penyimpanan data di lokasi pengguna	Konsistensi data antar node (CAP theorem)
Toleransi kegagalan (failover otomatis)	Biaya infrastruktur yang lebih tinggi
Mendukung beban kerja *read-heavy*	Keamanan data yang terdistribusi

2.4 Perbandingan dengan Caching Tradisional

Caching Tradisional	Distributed Caching
• Arsitektur single-node	• Multi-node terdistribusi
• Cocok untuk traffic rendah	• Skalabilitas elastis
• Risiko single point of failure	• Data di-replikasi di beberapa zona
• Contoh: Local Redis instance	• Contoh: Redis Cluster

Gambar 2.1: Perbandingan Arsitektur Caching Tradisional vs. Distributed

Ilustrasi Topologi Distributed Cache



Gambar 2.2: Contoh Topologi Distributed Cache dengan 3 Node

2.5 Prinsip Desain Penting

- **Sharding**: Memecah data ke beberapa node berdasarkan algoritma (misal: *consistent hashing*) untuk menghindari hotspot.
- **Replikasi**: Menyimpan salinan data di beberapa node untuk memastikan ketersediaan saat terjadi kegagalan.
- **Eviction Policy**: Mekanisme penghapusan data saat cache penuh (misal: LRU - Least Recently Used).

- **Discovery & Orchestration**: Alat seperti ZooKeeper atau etcd untuk mengelola keanggotaan node secara dinamis.

Best Practices

- **Pilih Model Konsistensi Sesuai Kebutuhan**: Gunakan *eventual consistency* untuk aplikasi toleran terhadap keterlambatan (misal: media sosial), dan *strong consistency* untuk sistem finansial.
- **Monitor Hit Ratio**: Targetkan hit ratio $> 90\%$ untuk memastikan cache efektif. Jika di bawah 70%, evaluasi strategi caching.
- **Gunakan Cache Layer Terisolasi**: Pisahkan layer cache dari logika bisnis untuk memudahkan scaling dan pemeliharaan.

3. Arsitektur Distributed Caching

3.1 Model Arsitektur

3.1.1 Sharded Cache

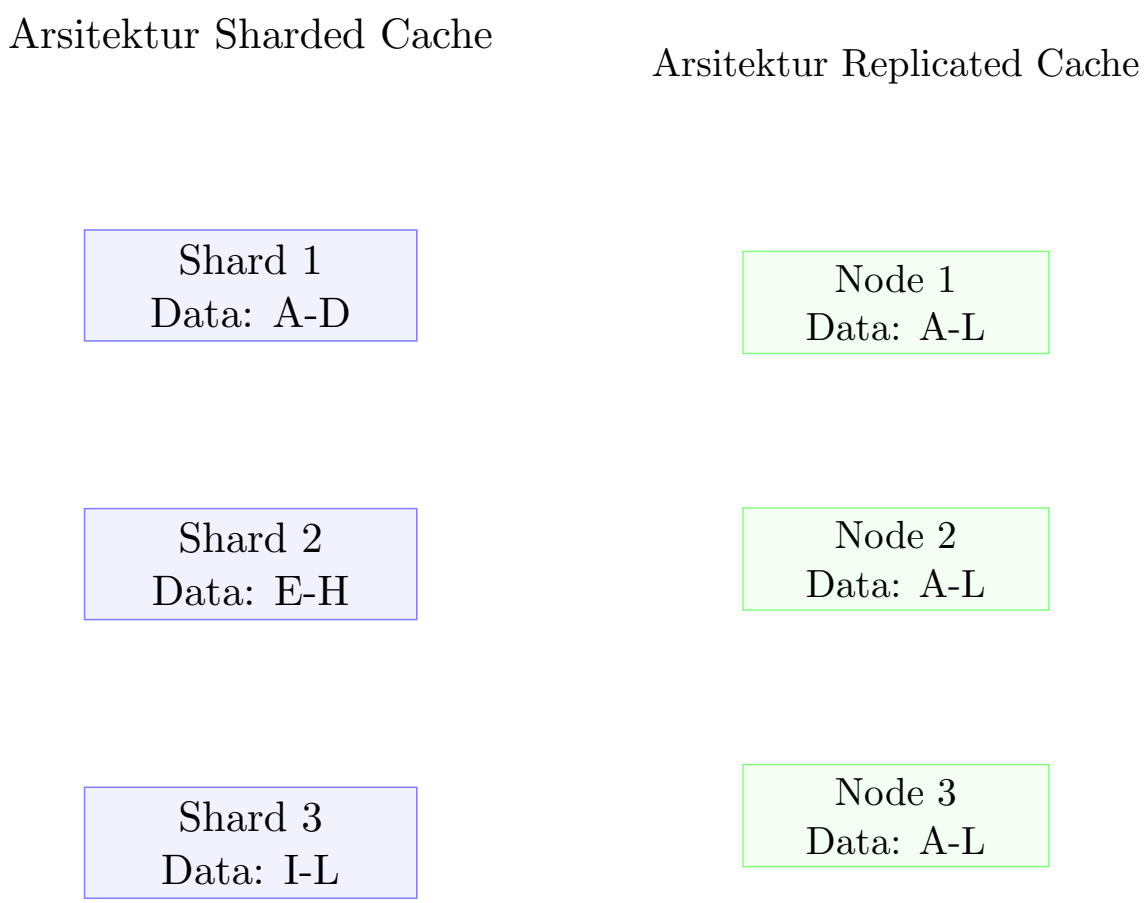
- **Konsep**: Data dipecah (*partitioned*) ke beberapa node menggunakan algoritma seperti *consistent hashing*. Setiap node hanya menyimpan subset data.
- **Contoh Implementasi**: Redis Cluster, Amazon ElastiCache.
- **Keuntungan**:
  - Skalabilitas horizontal tanpa batas.
  - Menghindari *hotspot* karena beban terdistribusi merata.
- **Kekurangan**:
  - Tidak ada replikasi otomatis (perlu konfigurasi tambahan).
  - Kompleksitas manajemen partisi.

3.1.2 Replicated Cache

- **Konsep**: Data diduplikasi ke semua node. Setiap node memiliki salinan lengkap cache.
- **Contoh Implementasi**: Hazelcast, Apache Ignite.
- **Keuntungan**:
  - Ketersediaan tinggi (*high availability*).
  - Akses data cepat karena tersedia di semua node.
- **Kekurangan**:
  - Penggunaan memori lebih tinggi.
  - Update data harus disinkronisasi ke semua node.

3.2 Strategi Konsistensi Data

- **Eventual Consistency**:
  - Data akan konsisten di semua node *akhirnya* (setelah delay tertentu).
  - Cocok untuk: media sosial, notifikasi non-kritis.
  - Contoh: Sistem like/comment di Facebook.



Gambar 3.1: Perbandingan Sharded vs. Replicated Cache

- **Strong Consistency**:
  - Data dijamin konsisten di semua node *sebelum* respons dikirim.
  - Cocok untuk: sistem finansial, reservasi tiket.
  - Contoh: Transaksi saham real-time.

Tabel 3.1: Perbandingan Strategi Konsistensi

Parameter	**Eventual Consistency**	**Strong Consistency**
Kecepatan	Tinggi	Rendah
Akurasi Data	Mungkin stale	Selalu akurat
Kompleksitas	Rendah	Tinggi
Use Case	Sosial media, IoT	Perbankan, E-commerce

3.3 Studi Kasus Arsitektur

3.3.1 Redis Cluster (Sharded)

- **Topologi**: 6 node (3 master + 3 replica).
- **Sharding**: Data dipecah menggunakan slot (total 16384 slot).
- **Keunikan**: Replikasi otomatis dari master ke replica.

3.3.2 Hazelcast (Replicated)

- **Topologi**: Peer-to-peer, semua node menyimpan data lengkap.
- **Sinkronisasi**: Multicast gossip protocol untuk update data.
- **Keunikan**: Auto-discovery node baru dalam jaringan.

Best Practices Pemilihan Arsitektur

- **Pilih Sharded Cache jika**: - Data sangat besar (>1 TB) - Traffic tidak merata (misal: *hot keys*)
- **Pilih Replicated Cache jika**: - Membutuhkan failover instan - Data relatif kecil (<100 GB)
- **Hindari Hybrid Model**: Kombinasi sharding + replikasi meningkatkan kompleksitas (kecuali menggunakan teknologi siap pakai seperti Redis Cluster).

4. Teknologi & Tools Populer

4.1 Redis

- **Deskripsi**: Redis adalah *in-memory data structure store* yang sangat populer, digunakan sebagai database, cache, dan message broker.
- **Fitur Unggulan**:
  - Dukungan struktur data kaya: string, hash, list, set, sorted set.
  - Pub/Sub untuk messaging real-time.
  - Replikasi otomatis dengan Redis Cluster.
  - Persistensi opsional (RDB snapshot dan AOF logging).
- **Use Case**:
  - Session storage untuk aplikasi web.
  - Leaderboard untuk aplikasi gaming.
  - Cache layer untuk database.
- **Keterbatasan**:
  - Single-threaded, sehingga performa terbatas pada beban CPU-intensive.
  - Memerlukan manajemen memori yang hati-hati karena data disimpan di RAM.

4.2 Memcached

- **Deskripsi**: Memcached adalah sistem caching terdistribusi yang sederhana dan cepat, dirancang untuk caching objek kecil.
- **Fitur Unggulan**:
  - Desain sederhana dengan overhead rendah.
  - Skalabilitas horizontal yang mudah.
  - Dukungan multi-threading.
- **Use Case**:
  - Caching fragment HTML untuk aplikasi web.
  - Penyimpanan sementara hasil query database.
- **Keterbatasan**:
  - Tidak mendukung struktur data kompleks (hanya key-value).
  - Tidak ada persistensi data.

4.3 DragonflyDB

- **Deskripsi**: DragonflyDB adalah *in-memory data store* yang kompatibel dengan Redis, dirancang untuk performa tinggi dan efisiensi CPU.
- **Fitur Unggulan**:
  - **Multi-threaded Architecture**: Mengatasi batasan Redis yang single-threaded dengan memanfaatkan CPU multi-core secara penuh.
  - **Tanpa Global Lock**: DragonflyDB menghilangkan *global lock*, sebuah mekanisme yang sering menjadi bottleneck pada sistem tradisional.
    - **Apa Itu Global Lock?** *Global lock* adalah mekanisme sinkronisasi yang memastikan hanya satu thread yang dapat mengakses data pada suatu waktu. Meskipun berguna untuk konsistensi, ini membatasi skalabilitas dan performa.
    - **Masalah Global Lock**: Pada sistem single-threaded seperti Redis, *global lock* menyebabkan thread lain harus menunggu, sehingga mengurangi throughput secara signifikan.
    - **Solusi DragonflyDB**: Dengan arsitektur tanpa *global lock*, DragonflyDB memungkinkan banyak thread bekerja secara paralel, meningkatkan throughput dan mengurangi latensi.
  - **Klaim 25x Throughput Lebih Tinggi**: Berdasarkan benchmark, DragonflyDB mampu menangani beban yang jauh lebih besar dibandingkan Redis.
- **Use Case**:
  - Migrasi dari Redis untuk aplikasi dengan beban tinggi.
  - Aplikasi yang memerlukan throughput ekstrem dengan latensi rendah.
- **Keterbatasan**:
  - Relatif baru, sehingga ekosistem dan komunitas belum sebesar Redis.
  - Dokumentasi dan tooling masih dalam pengembangan.

4.4 Hazelcast & Apache Ignite

- **Deskripsi**: Hazelcast dan Apache Ignite adalah solusi caching terdistribusi yang menawarkan komputasi in-memory dan caching untuk aplikasi enterprise.
- **Fitur Unggulan**:
  - Dukungan untuk komputasi terdistribusi (*distributed computing*).
  - Replikasi data otomatis dengan konsistensi yang dapat dikonfigurasi.
  - Integrasi dengan sistem enterprise seperti Hadoop dan Spark.
- **Use Case**:

  - Caching untuk aplikasi enterprise dengan kompleksitas tinggi.
  - Analisis data real-time dengan komputasi in-memory.

- **Keterbatasan**:
  - Lebih kompleks untuk diatur dan dikelola dibandingkan Redis atau Memcached.
  - Membutuhkan sumber daya yang lebih besar (CPU, memori).

4.5 Valkey

- **Deskripsi**: Valkey adalah fork open-source dari Redis yang dikembangkan oleh komunitas untuk memastikan keberlanjutan proyek setelah perubahan lisensi Redis. Valkey kompatibel dengan Redis dan fokus pada performa tinggi, stabilitas, dan skalabilitas.
- **Fitur Unggulan**:
  - **Redis-Compatible**: Mendukung semua perintah dan protokol Redis, memudahkan migrasi dari Redis.
  - **Multi-threaded Architecture**: Dirancang untuk memanfaatkan CPU multi-core secara optimal tanpa *global lock*.
  - **Peningkatan Performa**: Klaim peningkatan throughput hingga 3x dibanding Redis untuk operasi tertentu.
  - **Open Source dengan Komunitas Aktif**: Dikembangkan secara transparan oleh Linux Foundation dan kontributor independen.
- **Use Case**:
  - Pengganti Redis untuk organisasi yang memprioritaskan open source.
  - Aplikasi yang memerlukan skalabilitas tinggi dengan biaya lebih rendah.
- **Keterbatasan**:
  - Masih dalam tahap pengembangan awal.
  - Fitur enterprise (e.g., Redis Enterprise) belum tersedia.

Perbandingan Tools Populer

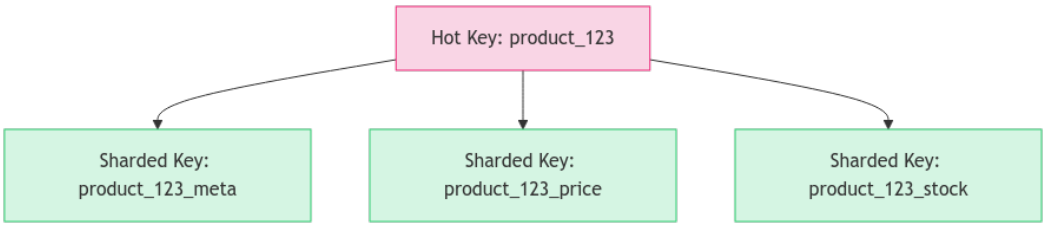
Tabel 4.1: Perbandingan Fitur Utama Tools Caching

Fitur	**Redis**	**Memcached**	**DragonflyDB**	**Valkey**	**Hazelcast/Ignite**
Lisensi	Sumber Terbatas (SSPL)	Open Source (BSD)	Sumber Terbatas	Open Source (BSD)	Open Source (Apache 2.0)
Kompatibilitas	-	-	Redis	Redis	-
Arsitektur	Single-threaded	Multi-threaded	Multi-threaded	Multi-threaded	Multi-threaded
Throughput	Tinggi	Sangat Tinggi	Sangat Tinggi (25x Redis)	Tinggi (3x Redis)	Sedang
Use Case Utama	Caching, Session, Messaging	Caching Objek Kecil	Migrasi Redis, High Throughput	Open Source Alternative	Enterprise, Komputasi Terdistribusi
Keterbatasan	Single-threaded	Struktur Data Terbatas	Ekosistem Belum Matang	Pengembangan Awal	Kompleksitas Tinggi

5. Best Practices Implementasi

5.1 Desain Kunci Cache

- **Hindari Hot Keys**: Hot keys terjadi ketika satu kunci cache diakses terlalu sering, menyebabkan beban berat pada satu node.
  - **Solusi**:
    - Bagi kunci besar menjadi bagian kecil (sharding)
    - Contoh: `product_123` → `product_123_meta`, `product_123_price`, `product_123_stock`
  - **Contoh Kasus**: Pada aplikasi e-commerce, kunci `flashsale_items` bisa di-shard menjadi `flashsale_items_1`, `flashsale_items_2`, dst.
- **Namespace untuk Organisasi Data**: Pisahkan data berdasarkan kategori untuk memudahkan manajemen.
  - Format: `<service>_<module>_<id>`
  - Contoh: `auth_session_user123`, `catalog_product_456`
  - Manfaat:
    - Mudah mencari dan mengelola data
    - Bisa menghapus data spesifik tanpa mempengaruhi yang lain
    - Mencegah tabrakan nama kunci



Gambar 5.1: Contoh Sharding Hot Key `product_123`

5.2 Manajemen TTL & Invalidation

5.2.1 Atur TTL Dinamis

- **Berdasarkan Pola Akses**:
  - Data yang sering diakses: TTL lebih panjang (misal: 1 jam).
  - Data jarang diakses: TTL lebih pendek (misal: 5 menit).
- **Berdasarkan Konteks Bisnis**:

  - Harga produk: TTL 1 menit (untuk update real-time).
  - Konten statis: TTL 24 jam.

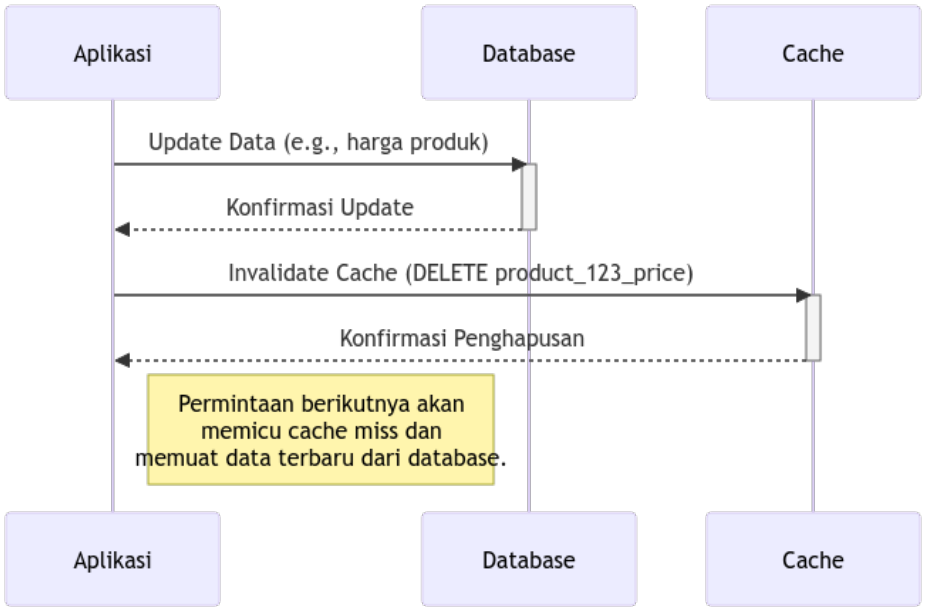
- **Implementasi Dinamis**: Gunakan algoritma adaptif seperti:

$$\text{TTL} = \text{Base\_TTL} + (\text{Hit\_Rate} \times \text{Adjustment\_Factor})$$

  Contoh: Jika hit rate 90%, TTL = 5 menit + (0.9 × 10 menit) = 14 menit.

5.2.2 Strategi Invalidation

- **Event-Driven Invalidation**:
  - Picu invalidasi cache saat ada perubahan di database (gunakan CDC tools seperti Debezium).
  - Contoh: Saat harga produk di-update, kirim event ke sistem cache untuk menghapus `product_<id>_price`.
- **Write-Through Cache**:
  - Setiap operasi tulis ke database langsung memperbarui cache.
  - Cocok untuk data kritis seperti stok barang.
- **Timeout-Based Invalidation**:
  - Gunakan TTL singkat untuk data yang sering berubah (misal: 10 detik).



Gambar 5.2: Alur Event-Driven Cache Invalidation

5.3 Monitoring & Observability

- **Metric Kunci**:
  - **Hit Ratio**: Rasio cache hit vs total request. Target: $> 90\%$.

$$\text{HitRatio} = \frac{\text{CacheHit}}{\text{CacheHit} + \text{CacheMiss}}$$

  - **Latensi**:
    - P95 latensi akses cache harus < 10 ms.
    - Gunakan histogram untuk analisis distribusi.
  - **Memory Usage**: Pantau penggunaan memori per node. Hindari $> 80\%$ kapasitas.
- **Tools Rekomendasi**:
  - Prometheus + Grafana untuk monitoring real-time.
  - Contoh Dashboard Grafana:
    - Panel hit ratio per service.
    - Heatmap latensi cache.
    - Alert saat memory usage $> 75\%$.

```yaml
scrape_configs:
  - job_name: 'redis'
    static_configs:
      - targets: ['redis-node1:9121', 'redis-node2:9121']
    metrics_path: /scrape
    relabel_configs:
      - source_labels: [__address__]
        target_label: instance
```

Listing 2: Contoh Konfigurasi Prometheus untuk Redis

5.4 Peringatan Penting

- **Cache Warming (Pemanasan Cache)**:
  - **Apa itu?**: Proses memuat data ke cache sebelum permintaan aktual terjadi, biasanya saat startup aplikasi atau sebelum event traffic tinggi.
  - **Mengapa penting?**: Menghindari cache miss massal yang menyebabkan lonjakan beban database.
  - **Cara Implementasi**:
    - Jalankan script inisialisasi yang mengakses endpoint kritis saat aplikasi mulai.
    - Contoh:

```
# Panggil API produk populer saat startup
curl https://api.example.com/products/top-100
```

Listing 5.1: Contoh Script Cache Warming

- **Cache Stampede**:
  - **Apa itu?**: Fenomena di mana banyak request bersamaan mencoba mengisi ulang cache yang sama saat data expired, menyebabkan overload pada database.
  - **Contoh Kasus**: 1000 request/detik untuk data produk yang TTL-nya habis bersamaan.
  - **Solusi**:
    - **Probabilistic Early Expiration**: Refresh cache secara acak sebelum TTL habis. Contoh:

$$\text{TTL\_aktual} = \text{TTL} - \text{random}(0, \text{TTL}/4)$$

    - **Locking Mechanism**: Gunakan distributed lock (e.g., Redis Redlock) untuk memastikan hanya satu request yang mengisi cache.
    - **Contoh Kode**:

```python
def get_data(key):
    data = cache.get(key)
    if not data:
        if acquire_lock(key):
            try:
                data = db.query_data(key)
                cache.set(key, data, ttl=300)
            finally:
                release_lock(key)
    return data
```

Listing 3: Penanganan Cache Stampede dengan Lock

Tabel 5.1: Kapan Harus dan Tidak Harus Menggunakan Cache

Kondisi	**Cache**	**Jangan Cache**
Frekuensi Akses	Tinggi (>100x/jam)	Rendah (<10x/jam)
Ukuran Data	Kecil (<1 MB)	Besar (>10 MB)
Volatilitas Data	Rendah (jarang berubah)	Tinggi (sering berubah)
Contoh	Halaman beranda, session user	Laporan PDF, data audit

6. Rangkuman & Rekomendasi

6.1 Poin Kunci

- **Distributed Caching Penting Untuk**:
  - Aplikasi dengan *high traffic* dan kebutuhan responsivitas tinggi.
  - Sistem yang memerlukan skalabilitas global dan toleransi kegagalan.
  - Mengurangi biaya operasional dengan menurunkan beban database.
- **Pemilihan Teknologi**:

Tabel 6.1: Pedoman Pemilihan Teknologi Caching Alternatif

Kebutuhan	**Rekomendasi Tool**
Kompatibilitas Redis + Open Source	Valkey
Throughput ekstrem + multi-threaded	DragonflyDB
Enterprise + komputasi terdistribusi	Hazelcast/Apache Ignite
Sederhana + caching objek kecil	Memcached

6.2 Resource Lanjutan

6.2.1 Dokumentasi Resmi

- **DragonflyDB**: https://dragonflydb.io/docs (Benchmark, konfigurasi klaster).
- **Redis**: https://redis.io/docs (Panduan struktur data, Redis Stack).
- **Valkey**: https://valkey.io/docs (Migrasi dari Redis, best practices).
- **Memcached**: https://memcached.org/documentation (Optimasi memori).
- **Hazelcast**: https://docs.hazelcast.com (Arsitektur enterprise).

6.2.2 Pembelajaran Lanjut

- **Redis University**: https://university.redis.com (Gratis - kursus "Redis for Java Developers").
- **Buku Direkomendasikan**:
  - *"Designing Data-Intensive Applications"* oleh Martin Kleppmann.
  - *"Redis in Action"* oleh Josiah Carlson.
- **Video Tutorial**:
  - YouTube: System Design Interview - Distributed Cache (34 menit).

  - Youtube: Redis vs Memcached Performance Benchmark
  - Youtube: Redis vs Dragonfly Performance (Latency - Throughput - Saturation)

Aksi Selanjutnya

- **Untuk Startup**: Mulai dengan Redis/Memcached + pantau hit ratio. Migrasi ke DragonflyDB jika mencapai 10k RPM+.
- **Untuk Enterprise**: Evaluasi Hazelcast/Apache Ignite untuk integrasi dengan sistem legacy.
- **Eksperimen Mandiri**:
  - Bandingkan latensi Redis vs DragonflyDB menggunakan `redis-benchmark`.
  - Implementasi cache warming di aplikasi Anda.

"Caching is like salt - the right amount enhances performance, but too much ruins the system."
- Analogi Arsitek Sistem