

**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**  
**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN**  
**KHOA CÔNG NGHỆ PHẦN MỀM**



# **Báo cáo**

# **Phương pháp mô hình hóa**

Giảng viên hướng dẫn  
**Ths. Nguyễn Công Hoan**

Sinh viên thực hiện  
**Phạm Nhật Huy – 23520643**  
**Hồ Nguyễn Tài Lợi – 23520869**  
**Nguyễn Trung Kiên – 23520802**  
**Nguyễn Tuấn Lộc – 23520862**

Thành phố Hồ Chí Minh, ngày 01/06/2025

# Mục lục

<b>Chương 1. System và software</b>	<b>11</b>
1.1. Giới thiệu	11
1.2. Thuộc tính chức năng và phi chức năng (functional requirements & non-functional requirements)	11
1.2.1. Thuộc tính chức năng (functional requirements)	11
1.2.1.1. Khái niệm	11
1.2.1.2. Vai trò và tầm quan trọng	11
1.2.1.3. Các ví dụ thực tế	13
1.2.2. Thuộc tính phi chức năng (non-functional requirements)	14
1.2.2.1. Khái niệm	14
1.2.2.2. Vai trò và tầm quan trọng	14
1.2.2.3. Các ví dụ thực tế	15
1.3. Lợi ích của cache trong system và software	16
1.3.1. Cải thiện hiệu năng (Performance Optimization)	16
1.3.1.1. Giảm thời gian phản hồi (Response Time)	16
1.3.1.2. Tăng thông lượng (Throughput)	17
1.3.2. Tối ưu Khả năng Xử lý Tải và Khả năng Mở rộng (Load Handling & Scalability)	17
1.3.2.1. Giảm tải cho cơ sở dữ liệu (Database Offloading)	17
1.3.2.2. Tăng khả năng mở rộng (Scalability)	17
1.3.3. Tăng cường Bảo mật (Enhanced Security)	17
1.3.3.1. Giảm mức độ phơi bày hệ thống backend (Backend Exposure Mitigation)	18
1.3.3.2. Bảo vệ dữ liệu người dùng (Data Privacy Compliance)	18
1.4. Cache trong google chrome	18
1.4.1. Cách thức Hoạt động của Browser Cache	18
1.4.1.1. Các loại tài nguyên được lưu trong cache:	18
1.4.1.2. Vị trí lưu trữ:	19
1.4.1.3. Cơ chế điều khiển:	19
1.4.2. Ví dụ Minh họa Cụ thể với Hình ảnh	19
1.4.2.1. Lần đầu truy cập một website:	19
1.4.2.2. Lần truy cập thứ hai:	19
1.4.2.3. Lợi ích rõ rệt từ Browser Cache:	19
1.4.3. Kiểm tra và Giám sát Cache trong Google Chrome	20
1.4.3.1. Các bước thực hiện:	20
<b>Chương 2. Tổng quan về cache</b>	<b>21</b>
2.1. Cache là gì?	21

2.2.	Lịch sử của cache .....	21
2.3.	Ưu điểm và nhược điểm của cache .....	24
2.3.1.	Ưu điểm của Cache .....	24
2.3.2.	Nhược điểm của Cache .....	27
2.4.	Các thông số và yếu tố hỗ trợ quan trọng thường sử dụng trong việc cache .....	29
2.5.	Cache invalidation .....	31
2.5.1.	Cloudflare: công nghệ và công cụ thực tiễn .....	31
2.5.1.1.	Purge cache (xóa cache thủ công) .....	31
2.5.1.2.	Cache-Tag (cache nhãn) .....	32
2.5.1.3.	Edge Cache TTL .....	32
2.5.1.4.	Browser Cache TTL .....	32
2.5.1.5.	Tái xác thực (Revalidation) .....	33
2.6.	Cache replacement policies .....	33
2.6.1.	Các chính sách nổi bật .....	34
2.6.1.1.	Thuật toán LRU (Least Recently Used) .....	35
2.6.1.2.	Thuật toán ARC (Adaptive Replacement Cache) .....	35
2.6.1.3.	Thuật toán thay thế ngẫu nhiên (Random replacement) .....	36
<b>Chương 3.</b>	<b>Các loại cache .....</b>	<b>37</b>
3.1.	Hardware Cache: .....	37
3.1.1.	CPU Cache: .....	37
3.1.1.1.	CPU memory cache là gì? .....	37
3.1.1.2.	Cấu trúc của bộ nhớ đệm CPU: .....	38
3.1.1.3.	Cơ chế hoạt động: .....	38
3.1.1.4.	Tầm quan trọng của bộ nhớ đệm CPU: .....	39
3.1.1.5.	Các công nghệ và xu hướng hiện đại: .....	40
3.1.2.	GPU cache: .....	41
3.1.2.1.	Bộ nhớ đệm GPU: .....	41
3.1.2.2.	Cấu trúc của bộ nhớ đệm CPU: .....	42
3.1.2.3.	Cơ chế hoạt động: .....	43
3.1.2.4.	Tầm quan trọng của bộ nhớ đệm GPU: .....	44
3.1.2.5.	Các công nghệ và xu hướng hiện đại: .....	45
3.2.	Disk Cache: .....	47
3.2.1.	Khái niệm: .....	47
3.2.2.	Cách disk cache hoạt động: .....	47
3.2.3.	Kiến trúc của disk cache: .....	47
3.2.4.	Tác dụng của disk cache: .....	48
3.2.4.1.	Tăng tốc độ truy cập dữ liệu: .....	48
3.2.4.2.	Cải thiện hiệu suất ghi: .....	48
3.2.4.3.	Tăng hiệu suất hệ thống tổng thể: .....	48
3.2.4.4.	Cải thiện trải nghiệm người dùng: .....	49

3.2.4.5.	Tăng độ an toàn dữ liệu (trong một số trường hợp):	49
3.2.5.	Tương lai của Disk Cache:	49
3.2.5.1.	Tăng hiệu suất với SSD và NVMe:	49
3.2.5.2.	Tích hợp bộ nhớ không bay hơi (NVM):	49
3.2.5.3.	Quản lý thông minh với AI và máy học:	50
3.2.5.4.	Hỗ trợ xử lý dữ liệu lớn (Big Data):	50
3.2.5.5.	Xu hướng mới:	50
3.3.	Network cache:	50
3.3.1.	Khái niệm:	50
3.3.2.	Phân loại:	51
3.3.3.	Chi tiết:	51
3.3.3.1.	DNS và DNS Cache:	51
3.3.3.2.	CDNcache:	57
3.4.	Software cache	60
3.4.1.	Các loại cache của hệ điều hành:	60
3.4.2.	Web cache:	61
3.4.2.1.	Cache của trình duyệt web (browser cache):	61
3.4.2.2.	HTTP Caching:	65
3.4.3.	Local Storage:	67
3.5.	Back-end:	69
3.5.1.	In-memory Cache:	69
3.5.2.	Các best practices cho Caching	73
3.6.	Write – Through Cache:	78
3.6.1.	Write-Through Cache là gì?	78
3.6.2.	Khi nào nên sử dụng Write-Through Cache?	78
3.6.3.	Cơ chế hoạt động của Write-Through Cache:	80
3.6.4.	Ưu và nhược điểm của Write-Through Cache:	81
3.6.5.	Các phương pháp để triển khai chiến lược Write – Through Cache	82
<b>Chương 4. Các phương pháp cache</b>		<b>84</b>
4.1.	Write aside (Lazy caching)	84
4.1.1.	Giới thiệu về caching	84
4.1.2.	Cách hoạt động	85
4.1.2.1.	Tổng quan về luồng thực hiện	85
4.1.2.2.	Chi tiết quá trình thực hiện	86
4.1.3.	Ưu và nhược điểm của Write aside (Lazy caching)	87
4.1.3.1.	Ưu điểm	87
4.1.3.2.	Nhược điểm	87
4.1.4.	Ví dụ cài đặt	87
4.1.4.1.	Mục tiêu	87
4.1.4.2.	Cấu trúc cơ bản	88

4.1.4.3.	Cài đặt .....	88
4.2.	Write-Through .....	91
4.2.1.	Giới thiệu Write-Through .....	91
4.2.2.	Hoạt động của bộ nhớ đệm write-through .....	91
4.2.3.	Mục đích bộ nhớ đệm write-through .....	92
4.2.4.	Ưu và nhược điểm của bộ nhớ đệm write-through .....	92
4.2.5.	Ví dụ triển khai .....	93
4.2.5.1.	Cấu trúc .....	93
4.2.5.2.	Maven Dependencies (giống write-aside) .....	93
4.2.5.3.	Entity .....	93
4.2.5.4.	Repository .....	94
4.2.5.5.	Service – Write-Through Cache Logic .....	94
4.2.5.6.	Controller .....	95
4.3.	Write behind (Write back) .....	96
4.3.1.	Giới thiệu Write-Behind Caching .....	96
4.3.2.	Ưu và nhược điểm của mô hình write-behind caching .....	97
4.3.2.1.	Ưu điểm .....	97
4.3.2.2.	Nhược điểm .....	97
4.3.3.	Các phương pháp để giảm tải cho cơ sở dữ liệu .....	98
4.3.3.1.	Sử dụng giới hạn tỷ lệ (rate limiting) .....	98
4.3.3.2.	Sử dụng kỹ thuật phân lô (batching) và hợp nhất (coalescing) .....	98
4.3.3.3.	Sử dụng chuyển dịch thời gian (time shifting) .....	98
4.3.4.	So sánh write-behind với write-through cache .....	98
4.3.5.	Ví dụ triển khai .....	99
4.3.5.1.	Mục tiêu: .....	99
4.3.5.2.	Cách triển khai .....	99
4.3.5.3.	Ý tưởng: .....	99
4.3.5.4.	Maven dependencies (giống các ví dụ trước) .....	99
4.3.5.5.	Entity & Repository .....	99
4.3.5.6.	Cache Store (dùng ConcurrentHashMap hoặc Redis) . . .	100
4.3.5.7.	Service: Write-Behind Logic .....	100
4.3.5.8.	Scheduled Flush Job (ghi DB định kỳ) .....	101
4.3.5.9.	Controller .....	102
4.4.	Read through .....	102
4.4.1.	Giới thiệu về Read-Through .....	102
4.4.2.	Cách hoạt động của mô hình read-through .....	103
4.4.3.	Ưu và nhược điểm của Read-through .....	103
4.4.3.1.	Ưu điểm .....	103
4.4.3.2.	Nhược điểm .....	104

4.4.4.	Read-Through so với Cache-Aside (Lazy cache)	105
4.4.5.	Ví dụ triển khai	105
4.4.5.1.	Mô tả	105
4.4.5.2.	Code ví dụ	105
4.4.5.3.	Kiểm thử	106
4.4.5.4.	Ghi chú	107
4.5.	Refresh ahead	107
4.5.1.	Giới thiệu Refresh Ahead	107
4.5.2.	Cách hoạt động của Refresh Ahead	108
4.5.3.	Ưu và nhược điểm của Refresh Ahead	108
4.5.3.1.	Ưu điểm	108
4.5.3.2.	Nhược điểm	109
4.5.3.3.	So sánh Refresh Ahead và Read-Through Cache	109
4.5.4.	Ví dụ triển khai	110
4.5.4.1.	Mô tả Refresh Ahead	110
4.5.4.2.	TypeScript với giả lập	110
4.5.4.3.	Cache có Refresh Ahead	110
4.5.4.4.	Sử dụng	111
4.5.4.5.	Lợi ích	111
4.6.	Pre-caching	112
4.6.1.	Hướng tiếp cận Pre-Caching	112
4.6.2.	Giới thiệu về Pre-caching	112
4.6.3.	Cách hoạt động của Pre-caching	113
4.6.4.	Sử dụng Pre-cache	114
4.6.5.	Ưu và nhược điểm của Pre-caching	114
4.6.5.1.	Ưu điểm	114
4.6.5.2.	Nhược điểm	115
4.6.6.	Phân loại Pre-caching	115
4.6.6.1.	Pre-caching phía máy khách	115
4.6.6.2.	Pre-caching phía máy chủ	116
4.6.7.	Tối ưu cho Pre-Caching	116
4.6.8.	Ví dụ triển khai	117
4.6.8.1.	Tạo Service Worker (service-worker.js)	117
4.6.8.2.	Đăng ký Service Worker trong File JavaScript Chính	118
4.6.8.3.	Cập nhật HTML	119
4.6.8.4.	Lưu ý	119
4.7.	Read-only cache	119
4.7.1.	Giới thiệu về Read-Only cache	119
4.7.2.	Đặc điểm chính	120
4.7.3.	Các ứng dụng và triển khai thực tế	121

4.7.3.1.	IBM – Local Read-Only Cache (LROC) .....	121
4.7.3.2.	Oracle – Read-Only Cache Group (TimesTen) .....	121
4.7.3.3.	Optimizely – Read-Only Object Cache .....	121
4.7.3.4.	Wikipedia – Page Cache (khái niệm hệ điều hành) .....	121
4.7.4.	Ưu và nhược điểm của Read-Only Cache .....	121
4.7.4.1.	Lợi ích của Read-Only Cache .....	121
4.7.4.2.	Hạn chế .....	122
4.7.5.	Ví dụ triển khai .....	122
4.8.	Negative caching .....	123
4.8.1.	Giới thiệu về Negative Caching .....	123
4.8.2.	Nguyên lý hoạt động .....	124
4.8.3.	Ưu và nhược điểm của Negative Caching .....	124
4.8.3.1.	Ưu điểm .....	124
4.8.4.	Ứng dụng của Negative Caching .....	125
4.8.4.1.	DNS Negative Caching .....	125
4.8.4.2.	CDN (Content Delivery Network) .....	125
4.8.4.3.	Bộ nhớ đệm ứng dụng Web .....	125
4.8.4.4.	Proxy Servers và Load Balancers .....	126
4.8.5.	Triển khai Negative Caching .....	126
4.8.5.1.	Các tham số cấu hình quan trọng .....	126
4.8.5.2.	Ví dụ triển khai .....	126
4.8.5.3.	Cài đặt trong các ngôn ngữ lập trình .....	127
4.8.6.	Thách thức và Cân nhắc khi triển khai .....	128
4.8.6.1.	Quản lý TTL .....	128
4.8.6.2.	Xử lý sự cố .....	128
4.8.6.3.	Tính nhất quán dữ liệu .....	128
4.8.7.	Best Practices và Khuyến nghị .....	129
4.8.7.1.	Sử dụng TTL phù hợp .....	129
4.8.7.2.	Phân biệt các loại lỗi .....	129
4.8.7.3.	Tối ưu hóa kích thước cache .....	129
4.8.7.4.	Kết hợp với các kỹ thuật khác .....	129
4.8.8.	Xu hướng và phát triển .....	129
4.8.8.1.	Edge Computing và Negative Caching .....	129
4.8.8.2.	Machine Learning trong Negative Caching .....	129
4.9.	Hybrid caching .....	130
4.9.1.	Giới thiệu về Hybrid Caching .....	130
4.9.1.1.	Bản chất của Hybrid Caching .....	131
4.9.1.2.	Sự cần thiết của Hybrid Caching .....	131
4.9.2.	Các kiểu Hybrid Caching phổ biến .....	131
4.9.2.1.	Multi-level Cache (Cache đa tầng) .....	131

4.9.2.2.	Distributed Hybrid Cache (Cache phân tán hỗn hợp) . .	132
4.9.2.3.	Write-Through và Write-Back Hybrid Cache . . . . .	134
4.9.2.4.	Multi-Algorithm Cache (Cache đa thuật toán) . . . . .	135
4.9.2.5.	Cache Tiering kết hợp In-Memory và Persistent Cache .	136
4.9.3.	Các thành phần của một hệ thống Hybrid Caching . . . . .	138
4.9.3.1.	Cache Storage (Nơi lưu trữ cache) . . . . .	138
4.9.3.2.	Cache Policies (Chính sách cache) . . . . .	139
4.9.3.3.	Cache Synchronization (Đồng bộ hóa cache) . . . . .	139
4.9.3.4.	Cache Monitoring và Analytics . . . . .	139
4.9.4.	Triển khai Hybrid Caching trong các ứng dụng thực tế . . . . .	140
4.9.4.1.	Hybrid Caching trong ứng dụng Web . . . . .	140
4.9.4.2.	Hybrid Caching trong Microservices . . . . .	141
4.9.5.	Ưu và nhược điểm của Hybrid Caching . . . . .	143
4.9.5.1.	Ưu điểm . . . . .	143
4.9.5.2.	Nhược điểm . . . . .	143
4.9.6.	Ứng dụng của Hybrid Caching . . . . .	143
4.10.	Graph Cache: Caching data in N Dimensional structures . . . . .	143
4.10.1.	Giới thiệu . . . . .	143
4.10.2.	Nguyên lý cơ bản của Graph Cache . . . . .	144
4.10.3.	So sánh với các hệ thống cache truyền thống . . . . .	145
4.10.4.	Kiến trúc của Graph Cache . . . . .	145
4.10.4.1.	Data Nodes (Nút dữ liệu) . . . . .	145
4.10.4.2.	Dimension Nodes (Nút chiều) . . . . .	145
4.10.4.3.	Edges (Cạnh) . . . . .	145
4.10.5.	Ứng dụng thực tế của Graph Cache . . . . .	145
4.10.5.1.	Hệ thống thương mại điện tử . . . . .	145
4.10.5.2.	Phân tích mạng xã hội . . . . .	145
4.10.5.3.	Hệ thống gợi ý (Recommendation Systems) . . . . .	146
4.10.6.	Triển khai Graph Cache . . . . .	146
4.10.7.	Thách thức và lưu ý khi sử dụng Graph Cache . . . . .	147
<b>Chương 5. Áp dụng cache vào microservice . . . . .</b>		<b>147</b>
5.1.	Giới thiệu microservice . . . . .	147
5.1.1.	Đặc điểm chính của Microservice . . . . .	148
5.1.2.	Lợi ích . . . . .	148
5.1.3.	Thách thức . . . . .	148
5.2.	Distributed cache . . . . .	149
5.2.1.	Giới thiệu . . . . .	149
5.2.2.	Định nghĩa Distributed Cache . . . . .	149
5.2.2.1.	Ví dụ các hệ thống Distributed Cache phổ biến: . . . . .	149
5.2.3.	Lợi ích của Distributed Cache trong Microservices . . . . .	150



5.2.4.	Kiến trúc Tích hợp Distributed Cache .....	150
5.2.4.1.	Cache-Aside (Lazy Loading) .....	150
5.2.4.2.	Write-Through / Write-Behind .....	150
5.2.5.	Vấn đề & Thách thức .....	150
5.2.6.	Best Practices .....	151
5.3.	Event-Driven Cache Invalidation .....	151
5.3.1.	1. Giới thiệu .....	151
5.3.2.	Vấn đề của Cache trong Microservice .....	151
5.3.3.	Event-Driven Cache Invalidation là gì? .....	151
5.3.4.	Quy trình hoạt động .....	152
5.3.5.	Ưu điểm .....	152
5.3.6.	Nhược điểm và thách thức .....	152
5.3.7.	Mô hình triển khai phổ biến .....	153
5.3.7.1.	Sơ đồ tổng quát: .....	153
5.3.8.	Ví dụ với Redis Cache và Kafka .....	153
5.4.	Distributed locking .....	153
5.4.1.	Giới thiệu .....	153
5.4.2.	Vấn đề đặt ra .....	153
5.4.3.	Định nghĩa Distributed Lock .....	154
5.4.4.	Một số cách triển khai Distributed Lock .....	154
5.4.4.1.	Dùng Redis (Redlock Algorithm) .....	154
5.4.4.2.	Dùng Zookeeper .....	154
5.4.4.3.	Dùng cơ chế cơ sở dữ liệu .....	155
5.4.5.	Một số thư viện hỗ trợ phổ biến .....	155
5.4.6.	Khi nào nên dùng Distributed Lock .....	155
5.5.	Cache access management (phân quyền giống như csdl) .....	155
5.5.1.	Giới thiệu .....	155
5.5.2.	Mô hình phân quyền truy cập cache .....	156
5.5.2.1.	Phân vùng namespace .....	156
5.5.2.2.	Token-based authentication .....	156
5.5.2.3.	Role-Based Access Control (RBAC) .....	157
5.5.2.4.	Attribute-Based Access Control (ABAC) .....	158
5.5.3.	Triển khai phân quyền cache .....	158
5.5.3.1.	Cache Proxy Pattern .....	158
5.5.3.2.	Tích hợp với API Gateway .....	159
5.5.3.3.	Sidecar Pattern .....	159
5.5.3.4.	Middleware trong Client Library .....	160
5.5.4.	Các công nghệ và công cụ hỗ trợ .....	160
5.5.4.1.	Redis ACL (Access Control List) .....	160
5.5.4.2.	AWS ElastiCache IAM Authentication .....	160

5.5.4.3.	HashiCorp Vault .....	161
5.5.4.4.	OPA (Open Policy Agent) .....	161
5.5.5.	Một số thách thức và giải pháp trong các khía cạnh .....	162
5.5.5.1.	Hiệu suất và độ trễ .....	162
5.5.5.2.	Vấn đề nhất quán trong phân tán .....	162
5.5.5.3.	Cache invalidation và quyền xóa .....	162
5.5.5.4.	Debugging và troubleshooting .....	162
5.6.	Scale cache (sharding, master-slave,...) .....	163
5.6.1.	Giới thiệu .....	163
5.6.2.	Các Chiến Lược Scale Cache trong Microservice .....	163
5.6.2.1.	Cache Sharding (Phân mảnh Cache) .....	163
5.6.2.2.	Master-Slave Replication .....	164
5.6.2.3.	Distributed Cache Clusters .....	165
5.6.2.4.	Cache-Aside Pattern trong Microservice .....	166
5.6.2.5.	Geo-distributed Caching .....	166
5.6.3.	Tối Ưu Hiệu Suất Cache .....	166
5.6.3.1.	Eviction Policies (Chính sách loại bỏ) .....	166
5.6.3.2.	Consistency Strategies (Chiến lược nhất quán) .....	167
5.6.3.3.	Monitoring và Analytics .....	167
5.6.4.	Các Công Nghệ Cache Phổ Biến cho Microservice .....	167
5.6.4.1.	Redis .....	167
5.6.4.2.	Memcached .....	167
5.6.4.3.	Hazelcast .....	167
5.6.4.4.	Couchbase .....	168
5.6.5.	Các Ví Dụ Thực Tế .....	168
5.6.5.1.	Ví dụ 1: Redis Cluster với Sharding .....	168
5.6.5.2.	Ví dụ 2: Consistent Hashing với Memcached .....	169
5.6.6.	Thách Thức và Giải Pháp .....	169
5.6.6.1.	Cache Invalidation .....	169
5.6.6.2.	Cache Stampede/Thundering Herd .....	169
5.6.6.3.	Quản Lý Bộ Nhớ .....	170
5.6.6.4.	Nhất Quán Dữ Liệu trong Môi Trường Phân Tán .....	170

# Chương 1. System và software

## 1.1. Giới thiệu

Trong quá trình phát triển phần mềm, việc xác định rõ các thuộc tính của hệ thống là vô cùng quan trọng để đảm bảo rằng sản phẩm đáp ứng được kỳ vọng của người dùng và vận hành hiệu quả trong môi trường thực tế. Các thuộc tính hệ thống được chia làm hai nhóm chính: thuộc tính chức năng (functional requirements) và thuộc tính phi chức năng (non-functional requirements). Cả hai đều đóng vai trò thiết yếu trong việc định hình kiến trúc, thiết kế và triển khai phần mềm.

## 1.2. Thuộc tính chức năng và phi chức năng (functional requirements & non-functional requirements)

### 1.2.1. Thuộc tính chức năng (functional requirements)

#### 1.2.1.1. Khái niệm

Thuộc tính chức năng là những yêu cầu mô tả các hành động hoặc chức năng cụ thể mà hệ thống phần mềm cần thực hiện để đáp ứng nhu cầu sử dụng của người dùng hoặc tổ chức. Những chức năng này thường liên quan trực tiếp đến các quy trình nghiệp vụ và nghiệp vụ cốt lõi mà hệ thống đang phục vụ. Chúng được xem là “xương sống” của phần mềm, là những tính năng người dùng có thể tương tác trực tiếp, như thao tác nhập liệu, tra cứu, chỉnh sửa, báo cáo, quản lý, xác thực đăng nhập...

#### 1.2.1.2. Vai trò và tầm quan trọng

Các yêu cầu chức năng (functional requirements) đóng một vai trò trung tâm và không thể thiếu trong toàn bộ vòng đời phát triển phần mềm. Chúng không chỉ là danh sách các tính năng, mà còn là kim chỉ nam định hình nên hệ thống, đảm bảo rằng sản phẩm cuối cùng thực sự đáp ứng được mục tiêu kinh doanh và nhu cầu của người dùng. Việc xác định và quản lý hiệu quả các yêu cầu này mang lại nhiều lợi ích chiến lược:

##### 1.2.1.2.1. Xác định Phạm vi Hệ thống Một Cách Rõ ràng :

Các yêu cầu chức năng chính là yếu tố then chốt để xác định phạm vi (scope) của dự án. Chúng vạch ra ranh giới rõ ràng về "cần làm gì" và "không cần làm gì" cho nhóm phát triển.

- Ngăn chặn mở rộng phạm vi (scope creep): Khi các chức năng được định nghĩa cụ thể, nguy cơ phát sinh thêm các tính năng không dự kiến trong quá trình phát triển sẽ giảm thiểu. Điều này giúp dự án đi đúng hướng, tránh lãng phí nguồn lực và thời gian.
- Tập trung nguồn lực: Nhóm phát triển có thể tập trung vào việc xây dựng những tính năng mang lại giá trị cao nhất cho người dùng và doanh nghiệp, thay vì phân tán vào các chức năng không cần thiết.

- Thiết lập kỳ vọng: Một phạm vi rõ ràng giúp thiết lập kỳ vọng thực tế cho tất cả các bên liên quan – từ nhà đầu tư, quản lý dự án cho đến người dùng cuối.

#### **1.2.1.2.2. Cơ sở Vững chắc cho Thiết kế và Phát triển :**

Yêu cầu chức năng là nền tảng mà từ đó toàn bộ quá trình thiết kế kiến trúc (architectural design) và phát triển mã nguồn (code development) được xây dựng.

- Hướng dẫn thiết kế: Các nhà thiết kế hệ thống sử dụng các yêu cầu này để hình dung cấu trúc tổng thể của phần mềm, cách các module tương tác với nhau và luồng dữ liệu. Chẳng hạn, yêu cầu "cập nhật đơn thuốc" sẽ dẫn đến việc thiết kế các bảng cơ sở dữ liệu liên quan đến thuốc, thông tin kê đơn, và các giao diện người dùng để nhập liệu.
- Chỉ đạo phát triển: Lập trình viên dựa vào các yêu cầu chức năng để viết mã nguồn. Mỗi chức năng được mô tả sẽ được chuyển thành một hoặc nhiều phần của mã lệnh, các lớp (classes), hàm (functions) hoặc component cụ thể.
- Kiểm thử và Đảm bảo chất lượng: Yêu cầu chức năng cung cấp tiêu chí để kiểm thử (testing). Mỗi yêu cầu đều có thể được kiểm tra độc lập để xác minh rằng hệ thống hoạt động đúng như mong đợi. Điều này là nền tảng cho việc đảm bảo chất lượng phần mềm trước khi triển khai.

#### **1.2.1.2.3. Liên kết Trực tiếp với Người dùng Cuối**

Đây là khía cạnh quan trọng nhất về tầm quan trọng của yêu cầu chức năng. Chúng là phần người dùng cuối sẽ trực tiếp tiếp xúc và sử dụng.

- Trải nghiệm người dùng (UX): Nếu các chức năng không đáp ứng được nhu cầu hoặc khó sử dụng, dù hệ thống có được xây dựng với công nghệ tiên tiến đến đâu, nó cũng sẽ thất bại trong việc được chấp nhận và sử dụng rộng rãi.
- Sự hài lòng của người dùng: Các chức năng được xây dựng đúng và đủ sẽ trực tiếp giải quyết các vấn đề, nâng cao hiệu suất làm việc và mang lại sự hài lòng cho người dùng. Ngược lại, việc thiếu đi một chức năng thiết yếu hoặc một chức năng bị lỗi có thể gây ra sự thất vọng và cản trở công việc hàng ngày.

#### **1.2.1.2.4. Tiết kiệm Chi phí và Tránh Rủi ro :**

Việc đầu tư thời gian và công sức vào giai đoạn thu thập và phân tích yêu cầu chức năng một cách kỹ lưỡng ngay từ đầu sẽ mang lại lợi ích tài chính đáng kể.

- Giảm thiểu sửa đổi lớn: Phát hiện lỗi hoặc thiếu sót ở giai đoạn đầu (phân tích yêu cầu) sẽ rẻ hơn rất nhiều so với việc sửa chữa chúng ở giai đoạn cuối (sau khi đã triển khai). Một thay đổi nhỏ ở bản vẽ thiết kế sẽ không tốn kém bằng việc phải đập bỏ và xây lại một phần của công trình đã hoàn thành.
- Tránh phát sinh chi phí không mong muốn: Sự mơ hồ trong yêu cầu có thể dẫn đến việc phát triển các tính năng không cần thiết hoặc hiểu sai mục đích, gây lãng phí nguồn lực và thời gian.

- Giảm thiểu rủi ro dự án: Các yêu cầu chức năng rõ ràng giúp quản lý rủi ro tốt hơn, vì nhóm có thể dự đoán và lập kế hoạch cho các thách thức tiềm ẩn liên quan đến việc triển khai từng chức năng.

### **1.2.1.3. Các ví dụ thực tế**

#### **1.2.1.3.1. Quản lý Hồ sơ Bệnh nhân và Lướt khám**

Cho phép nhân viên y tế tạo hồ sơ khám mới cho bệnh nhân:

Yêu cầu này là nền tảng cho mọi hoạt động khám chữa bệnh. Khi một bệnh nhân mới đến phòng khám hoặc bệnh viện, nhân viên y tế (ví dụ: lễ tân, điều dưỡng) cần có khả năng nhanh chóng và dễ dàng nhập thông tin để lập một hồ sơ cá nhân. Điều này bao gồm các thông tin cơ bản như Họ tên đầy đủ, Ngày sinh, Giới tính, Địa chỉ liên hệ, Số điện thoại, và nếu có, Mã số bảo hiểm y tế (BHYT) hoặc các thông tin bảo hiểm khác. Hệ thống cần đảm bảo tính toàn vẹn dữ liệu, ví dụ như cảnh báo nếu có trùng lặp thông tin bệnh nhân để tránh tạo nhiều hồ sơ cho cùng một người. Việc tạo hồ sơ mới cũng đồng nghĩa với việc khởi tạo một mã bệnh nhân duy nhất để dễ dàng tra cứu và quản lý trong tương lai.

#### **1.2.1.3.2. Tra cứu và Quản lý Danh mục**

Hiển thị danh sách loại thuốc, loại bệnh và cho phép tìm kiếm nhanh: Để hỗ trợ hiệu quả công việc của bác sĩ và dược sĩ, phần mềm cần có khả năng hiển thị các danh mục tham chiếu quan trọng.

Danh sách loại thuốc: Hệ thống phải hiển thị đầy đủ thông tin về các loại thuốc có sẵn trong kho, bao gồm tên thuốc (biệt dược và gốc), hoạt chất chính, đơn vị tính (viên, ml, gói), giá bán, và đặc biệt là số lượng tồn kho hiện tại. Việc hiển thị này cần được tổ chức khoa học, có thể theo nhóm thuốc hoặc dạng bào chế.

Danh sách loại bệnh (ICD-10): Phần mềm cần tích hợp và hiển thị danh mục các mã bệnh theo chuẩn ICD-10 (International Classification of Diseases, 10th Revision). Mỗi mã bệnh phải đi kèm với mô tả chi tiết để bác sĩ dễ dàng lựa chọn và ghi nhận chẩn đoán chính xác. Chức năng tìm kiếm nhanh: Điều quan trọng là phải có một cơ chế tìm kiếm mạnh mẽ cho cả danh mục thuốc và danh mục bệnh. Người dùng cần có thể tìm kiếm theo tên, mã, hoặc các từ khóa liên quan để nhanh chóng định vị thông tin cần thiết, giúp tiết kiệm thời gian và giảm thiểu sai sót.

#### **1.2.1.3.3. Thống kê và Báo cáo**

Thống kê số lượt khám theo từng loại bệnh trong một tháng:

Yêu cầu này phục vụ cho mục đích quản lý và phân tích nghiệp vụ. Hệ thống cần có khả năng tổng hợp dữ liệu từ các lượt khám và cung cấp các báo cáo thống kê. Ví dụ, việc thống kê số lượt khám theo từng loại bệnh trong một khoảng thời gian (ví dụ: một tháng, một quý) giúp ban lãnh đạo phòng khám/bệnh viện:

Nắm bắt xu hướng bệnh tật trong cộng đồng. Đánh giá hiệu quả các chương trình y tế dự phòng. Phân bổ nguồn lực (ví dụ: nhân sự, thuốc men) phù hợp hơn cho các chuyên khoa có nhu cầu cao. Đưa ra các quyết định chiến lược về mở rộng dịch vụ hoặc tập trung vào các lĩnh vực nhất định. Báo cáo này cần có thể được xuất ra dưới nhiều định dạng (ví dụ: Excel, PDF) và có khả năng lọc theo các tiêu chí khác nhau (ví dụ: theo bác sĩ, theo khoa).

## **1.2.2. Thuộc tính phi chức năng (non-functional requirements)**

### **1.2.2.1. Khái niệm**

Thuộc tính phi chức năng là những yêu cầu không liên quan trực tiếp đến các chức năng cụ thể mà hệ thống phải thực hiện, mà mô tả cách thức mà hệ thống đó vận hành, hiệu suất, độ tin cậy, khả năng mở rộng, bảo mật, và tính thân thiện với người dùng.

Khác với các thuộc tính chức năng (trả lời câu hỏi “hệ thống làm gì”), các thuộc tính phi chức năng trả lời cho câu hỏi “hệ thống hoạt động như thế nào”. Chúng ảnh hưởng sâu sắc đến trải nghiệm người dùng, tính hiệu quả khi triển khai, và tính duy trì về lâu dài của hệ thống phần mềm.

### **1.2.2.2. Vai trò và tầm quan trọng**

Mặc dù thường ít được chú trọng hơn so với các yêu cầu chức năng trong giai đoạn đầu phát triển phần mềm, nhưng các yêu cầu phi chức năng có tác động trực tiếp đến chất lượng tổng thể, tính ổn định, và khả năng duy trì của hệ thống. Nếu không được thiết kế và kiểm soát đúng cách, hệ thống có thể vẫn “hoạt động được” nhưng kém hiệu quả, khó mở rộng, dễ bị tấn công, hoặc gây khó chịu cho người dùng.

#### **1.2.2.2.1. Nâng cao Trải nghiệm Người dùng (UX)**

- **Tốc độ phản hồi nhanh:** Giảm thiểu độ trễ trong các thao tác giúp người dùng không cảm thấy chờ đợi, đặc biệt quan trọng trong các hệ thống thời gian thực như phần mềm y tế hoặc quản lý hàng tồn kho.
- **Giao diện mượt mà, nhất quán:** Một hệ thống có tính nhất quán trong giao diện và phản hồi sẽ tạo cảm giác tin cậy và dễ sử dụng, góp phần tăng cường sự hài lòng và mức độ chấp nhận của người dùng.

#### **1.2.2.2.2. Đảm bảo Khả năng Mở rộng và Bảo trì**

- **Khả năng mở rộng (Scalability):** Hệ thống cần được thiết kế để dễ dàng nâng cấp – từ việc xử lý thêm người dùng, dữ liệu lớn hơn, đến tích hợp với các hệ thống khác trong tương lai mà không cần tái thiết kế toàn bộ.
- **Tính mô-đun và dễ bảo trì:** Hệ thống phải có cấu trúc rõ ràng để dễ dàng cập nhật, vá lỗi hoặc bổ sung tính năng mà không ảnh hưởng đến toàn bộ hoạt động.

#### **1.2.2.2.3. Đáp ứng Khả năng Xử lý Tải và Hiệu suất**

- Throughput (Lưu lượng xử lý): Hệ thống phải có khả năng xử lý khối lượng lớn dữ liệu hoặc lượng truy cập đồng thời mà không bị gián đoạn hoặc chậm trễ.
- Response Time (Thời gian phản hồi): Mỗi hành động của người dùng nên được phản hồi trong thời gian chấp nhận được – thường dưới 2 giây trong các ứng dụng giao diện đồ họa.
- Tối ưu tài nguyên (CPU, RAM, I/O): Hệ thống cần tiết kiệm tài nguyên hệ thống, đặc biệt khi triển khai trên các máy chủ có giới hạn phần cứng.

#### **1.2.2.2.4. Tăng cường An toàn và Bảo mật**

- Xác thực và phân quyền: Đảm bảo chỉ những người dùng hợp lệ mới có thể truy cập hệ thống và chỉ có quyền tương ứng với vai trò của họ.
- Mã hóa và bảo vệ dữ liệu: Dữ liệu bệnh nhân, đơn thuốc hay lịch sử khám bệnh là các thông tin nhạy cảm cần được bảo mật nghiêm ngặt để tuân thủ các chuẩn mực như HIPAA hoặc các quy định quốc gia về bảo vệ dữ liệu cá nhân.
- Ghi log và phát hiện truy cập trái phép: Hệ thống cần có khả năng ghi nhận các hành động nghi vấn và hỗ trợ điều tra khi xảy ra sự cố bảo mật.

#### **1.2.2.2.5. Tăng Độ Tin Cậy và Tính Sẵn Sàng**

- Tính ổn định (Reliability): Hệ thống phải hoạt động ổn định trong thời gian dài mà không bị lỗi hoặc sập đột ngột.
- Khả năng phục hồi (Recoverability): Trong trường hợp hệ thống bị lỗi hoặc sự cố, cần có cơ chế sao lưu và khôi phục dữ liệu nhanh chóng.
- High Availability (Sẵn sàng cao): Đảm bảo hệ thống luôn hoạt động, đặc biệt trong các môi trường yêu cầu thời gian hoạt động 24/7 như phòng khám, bệnh viện.

### **1.2.2.3. Các ví dụ thực tế**

#### **1.2.2.3.1. Thời gian phản hồi khi thao tác tìm kiếm thuốc**

Hệ thống cần đảm bảo rằng khi người dùng gõ tên thuốc vào ô tìm kiếm, danh sách kết quả hiển thị gần như ngay lập tức (ví dụ dưới 1 giây). Điều này rất quan trọng trong môi trường khám bệnh có nhịp độ nhanh. Nếu thời gian phản hồi quá lâu (vài giây), bác sĩ hoặc dược sĩ sẽ bị gián đoạn quy trình, dễ dẫn đến sai sót hoặc bức xúc.

Để đáp ứng yêu cầu này, phần mềm nên sử dụng các kỹ thuật tối ưu như: cache bộ nhớ, tìm kiếm theo từ khóa không dấu, chỉ lấy các trường dữ liệu cần thiết, và dùng cơ chế truy vấn bất đồng bộ (asynchronous).

#### **1.2.2.3.2. Tính ổn định và sẵn sàng cao trong giờ cao điểm**

Vào giờ cao điểm (ví dụ: 7h30–9h00 sáng), có thể có hàng chục người dùng đăng nhập và sử dụng hệ thống cùng lúc. Phần mềm cần đảm bảo hoạt động ổn định, không bị treo hoặc phản hồi chậm. Nếu hệ thống bị sập giữa giờ khám bệnh, hậu quả có thể nghiêm trọng – gây gián đoạn tiếp nhận bệnh nhân, thất thoát dữ liệu, hoặc sai sót trong kê đơn.

Việc này đòi hỏi hệ thống được kiểm thử tải (load testing) kỹ càng trước khi triển khai, đồng thời có các cơ chế dự phòng như sao lưu tự động và ghi log lỗi để phục hồi nhanh chóng nếu gặp sự cố.

#### **1.2.2.3.3. Bảo mật dữ liệu bệnh nhân**

Thông tin bệnh nhân (họ tên, địa chỉ, tình trạng bệnh, thuốc đã dùng) là dữ liệu nhạy cảm và phải được bảo mật tuyệt đối. Hệ thống cần thực hiện:

- Cơ chế đăng nhập với mật khẩu mã hóa (bcrypt, SHA-256,...).
- Phân quyền rõ ràng: Nhân viên lễ tân không thể xem đơn thuốc, bác sĩ không thể chỉnh sửa thông tin hệ thống,...
- Mã hóa dữ liệu khi lưu trữ và khi truyền tải (sử dụng HTTPS, SSL).
- Ghi nhận lịch sử truy cập: Ai đăng nhập lúc nào, thao tác gì.

Điều này giúp phòng chống rò rỉ thông tin và đáp ứng các tiêu chuẩn về bảo mật dữ liệu trong lĩnh vực y tế.

### **1.3. Lợi ích của cache trong system và software**

#### **1.3.1. Cải thiện hiệu năng (Performance Optimization)**

Mục tiêu: Đảm bảo hệ thống phản hồi nhanh, ổn định và đáp ứng tốt khối lượng truy cập lớn.

##### **1.3.1.1. Giảm thời gian phản hồi (Response Time)**

Để tăng tốc độ phản hồi khi người dùng tương tác với phần mềm (ví dụ: tìm kiếm thuốc, truy xuất danh sách bệnh nhân), hệ thống cần áp dụng cơ chế cache thông minh. Việc lưu trữ tạm thời những dữ liệu được truy cập thường xuyên (chẳng hạn: danh sách loại thuốc, danh mục ICD-10, danh sách bác sĩ...) ngay tại lớp trung gian hoặc gần phía người dùng (edge caching) giúp:

- Giảm thời gian truy xuất dữ liệu từ vài giây xuống chỉ còn vài milliseconds.
- Tăng tốc độ tải giao diện và rút ngắn thời gian thao tác của người dùng, đặc biệt trong các quy trình nhiều bước (ví dụ: tạo đơn thuốc).
- Cải thiện trải nghiệm tổng thể, đặc biệt trong môi trường bệnh viện/phòng khám nơi mỗi giây đều quan trọng.



### **1.3.1.2. Tăng thông lượng (Throughput)**

Ngoài tốc độ phản hồi, hệ thống cũng cần xử lý được nhiều yêu cầu đồng thời hơn trong cùng một khoảng thời gian. Việc tối ưu throughput mang lại các lợi ích sau:

- Cho phép nhiều người dùng (lễ tân, bác sĩ, dược sĩ...) thao tác cùng lúc mà không bị chậm trễ.
- Giảm tải trực tiếp cho các backend services như cơ sở dữ liệu hoặc máy chủ xử lý nghiệp vụ.
- Góp phần giảm nguy cơ tắc nghẽn hệ thống, đặc biệt trong giờ cao điểm (7h30 – 9h sáng, hoặc sau giờ trưa).

### **1.3.2. Tối ưu Khả năng Xử lý Tải và Khả năng Mở rộng (Load Handling & Scalability)**

#### **1.3.2.1. Giảm tải cho cơ sở dữ liệu (Database Offloading)**

Việc triển khai lớp cache hiệu quả có thể giúp giảm đáng kể số lượng truy vấn gửi đến cơ sở dữ liệu. Cụ thể:

- Dữ liệu đọc nhiều (read-heavy) như danh sách danh mục, lịch sử đơn thuốc,... nên được cache tại lớp trung gian (middleware).
- Tỷ lệ cache hit cao sẽ giúp giảm số lần truy cập database thực tế, nhờ đó:
  - Cải thiện hiệu suất tổng thể của cơ sở dữ liệu.
  - Cho phép phục vụ nhiều người dùng đồng thời hơn.
  - Tránh tình trạng quá tải dẫn đến lỗi hoặc mất ổn định (database overload/crash).

#### **1.3.2.2. Tăng khả năng mở rộng (Scalability)**

Trong tương lai, khi số lượng người dùng tăng hoặc khi triển khai hệ thống tại nhiều cơ sở khám chữa bệnh, phần mềm cần có khả năng mở rộng linh hoạt:

- Horizontal scaling: Thêm nhiều node cache (cache cluster) giúp mở rộng hệ thống một cách hiệu quả về chi phí (cost-effective), mà không phải nâng cấp phần cứng hiện tại.
- Load balancer có thể phân phối truy cập đến các nút xử lý dữ liệu và cache khác nhau.
- Việc sử dụng các công nghệ phân tán (như Redis Cluster, Memcached Distributed) cũng giúp đảm bảo độ tin cậy và hiệu suất cao khi mở rộng.

### **1.3.3. Tăng cường Bảo mật (Enhanced Security)**

Yêu cầu bảo mật dữ liệu y tế là cực kỳ quan trọng, do tính chất nhạy cảm của thông tin liên quan đến bệnh nhân.

### 1.3.3.1. Giảm mức độ phơi bày hệ thống backend (Backend Exposure Mitigation)

Lớp cache không chỉ đóng vai trò tăng hiệu năng, mà còn có tác dụng như một lớp buffer bảo vệ hệ thống backend:

- Giảm lượng kết nối trực tiếp đến cơ sở dữ liệu, từ đó hạn chế nguy cơ bị tấn công từ bên ngoài (SQL injection, brute force query,...).
- Một số loại cache (như CDN hoặc reverse proxy) có thể lọc và chặn các truy cập bất thường hoặc độc hại, đóng vai trò như tường lửa ứng dụng (Application Firewall).
- Hạn chế khả năng bị tấn công từ chối dịch vụ (DDoS) bằng cách hấp thụ lượng truy cập lớn vào cache thay vì để hệ thống backend xử lý trực tiếp.

### 1.3.3.2. Bảo vệ dữ liệu người dùng (Data Privacy Compliance)

Dữ liệu y tế thường thuộc diện bảo mật cao (theo luật pháp và quy định ngành y tế), nên cần đặc biệt lưu ý trong quá trình cache:

- Không lưu trữ dữ liệu nhạy cảm (PII/PHI) như chẩn đoán, số CMND/BHYT, địa chỉ... trong bộ nhớ cache, trừ khi đã được mã hóa hoặc có biện pháp kiểm soát.
- Thiết lập cache expiration (TTL) phù hợp để đảm bảo dữ liệu không bị lưu trữ lâu hơn mức cần thiết.
- Mã hóa nội dung cache nếu có khả năng truy cập từ nhiều nguồn.
- Thực hiện kiểm tra và audit định kỳ các thành phần cache để đảm bảo tuân thủ quy định bảo mật (như HIPAA, **Nghị định 13/2023/NĐ-CP** tại Việt Nam).

## 1.4. Cache trong google chrome

Trong quá trình tối ưu hiệu năng hệ thống, một trong những kỹ thuật hiệu quả là tận dụng cơ chế cache trình duyệt (browser cache). Đây là một phương pháp giúp cải thiện tốc độ tải trang, giảm tải cho máy chủ, và mang lại trải nghiệm người dùng tốt hơn, đặc biệt với các ứng dụng có giao diện web như phần mềm quản lý phòng khám hoặc bệnh viện.

### 1.4.1. Cách thức Hoạt động của Browser Cache

Browser cache là cơ chế lưu trữ tạm thời các tài nguyên tĩnh (static assets) của một website hoặc ứng dụng web ngay trên máy người dùng, nhằm tái sử dụng các tài nguyên này trong các lần truy cập sau.

Đối với trình duyệt **Google Chrome** – một trong những trình duyệt phổ biến nhất hiện nay – cơ chế hoạt động của browser cache bao gồm:

#### 1.4.1.1. Các loại tài nguyên được lưu trong cache:

- Hình ảnh (images): .png, .jpg, .svg, v.v.
- Tập định dạng giao diện: .css
- Mã JavaScript (bao gồm logic xử lý phía client)
- Các trang HTML và font chữ

#### 1.4.1.2. Vị trí lưu trữ:

- Chrome lưu cache trong thư mục cục bộ (local cache folder) của hệ điều hành.
- Các tài nguyên này được quản lý và truy xuất tự động bởi trình duyệt.

#### 1.4.1.3. Cơ chế điều khiển:

Việc cache hay không, cache trong bao lâu, và khi nào cần làm mới cache được điều khiển thông qua các HTTP response headers, ví dụ:

- Cache-Control: Quy định thời gian lưu cache (ví dụ: max-age=31536000)
- ETag: Xác định phiên bản tài nguyên, hỗ trợ cache validation
- Expires: Thời điểm hết hạn của tài nguyên cache

Khi người dùng truy cập một trang web lần đầu, trình duyệt sẽ tải toàn bộ tài nguyên từ máy chủ. Những lần truy cập sau đó, trình duyệt sẽ kiểm tra bộ nhớ cache trước để xác định có thể sử dụng lại dữ liệu hay không, từ đó giảm số lượng yêu cầu gửi lên server.

#### 1.4.2. Ví dụ Minh họa Cụ thể với Hình ảnh

Để minh họa rõ ràng cơ chế cache trình duyệt, ta xét một tình huống cụ thể:

##### 1.4.2.1. Lần đầu truy cập một website:

- Người dùng mở trang web lần đầu tiên.
- Chrome gửi yêu cầu (HTTP request) đến server để tải các thành phần như:
  - Hình ảnh logo
  - File CSS, JS, ảnh nền,...
- Ví dụ: Tập hình ảnh logo.png (500KB) được tải về từ server.
- Chrome lưu hình ảnh này vào bộ nhớ cache local.
- Thời gian tải: khoảng 2–3 giây tùy tốc độ mạng.

##### 1.4.2.2. Lần truy cập thứ hai:

- Khi người dùng truy cập lại cùng website:
  - Trình duyệt kiểm tra bộ nhớ cache.
  - Nếu logo.png vẫn còn hiệu lực:
    - Trình duyệt tải hình ảnh từ cache, không cần truy cập server.
- Thời gian tải: chỉ khoảng 50–100 milliseconds.

##### 1.4.2.3. Lợi ích rõ rệt từ Browser Cache:

- Hiệu năng (Performance):
  - Tốc độ tải trang tăng lên rõ rệt.
  - Các thành phần giao diện có thể hiển thị tức thì.
  - Cache giúp giảm thời gian tải trang gấp 20–30 lần cho tài nguyên tĩnh.

- Tiết kiệm tài nguyên:
  - Giảm lưu lượng mạng (Bandwidth): Không cần tải lại hình ảnh, JS, CSS.
  - Giảm tải cho server backend: Ít truy vấn lặp lại không cần thiết.
- Trải nghiệm người dùng (UX):
  - Phản hồi nhanh, giao diện mượt mà.
  - Giảm thời gian chờ đợi.
  - Rất hữu ích trong môi trường như bệnh viện/phòng khám – nơi yêu cầu tốc độ và độ chính xác cao.

### 1.4.3. Kiểm tra và Giám sát Cache trong Google Chrome

Các nhà phát triển có thể dễ dàng xem và phân tích cache của trình duyệt bằng công cụ Chrome DevTools.

#### 1.4.3.1. Các bước thực hiện:

##### 1. Mở Chrome DevTools:

- Nhấn F12 hoặc chuột phải > Inspect.

##### 2. Tab Network:

- Chọn tab Network.
- Tick vào ô "Disable cache" để thử tải trang mà không dùng cache.
- Reload lại trang để kiểm tra:
  - Cột Status:
    - 200: tải mới từ server
    - 304 Not Modified: dùng cache cũ, không cần tải lại
    - (from disk cache) / (from memory cache): tài nguyên lấy từ cache
  - Cột Size: hiển thị rõ nguồn gốc của tài nguyên.

##### 3. Tab Application:

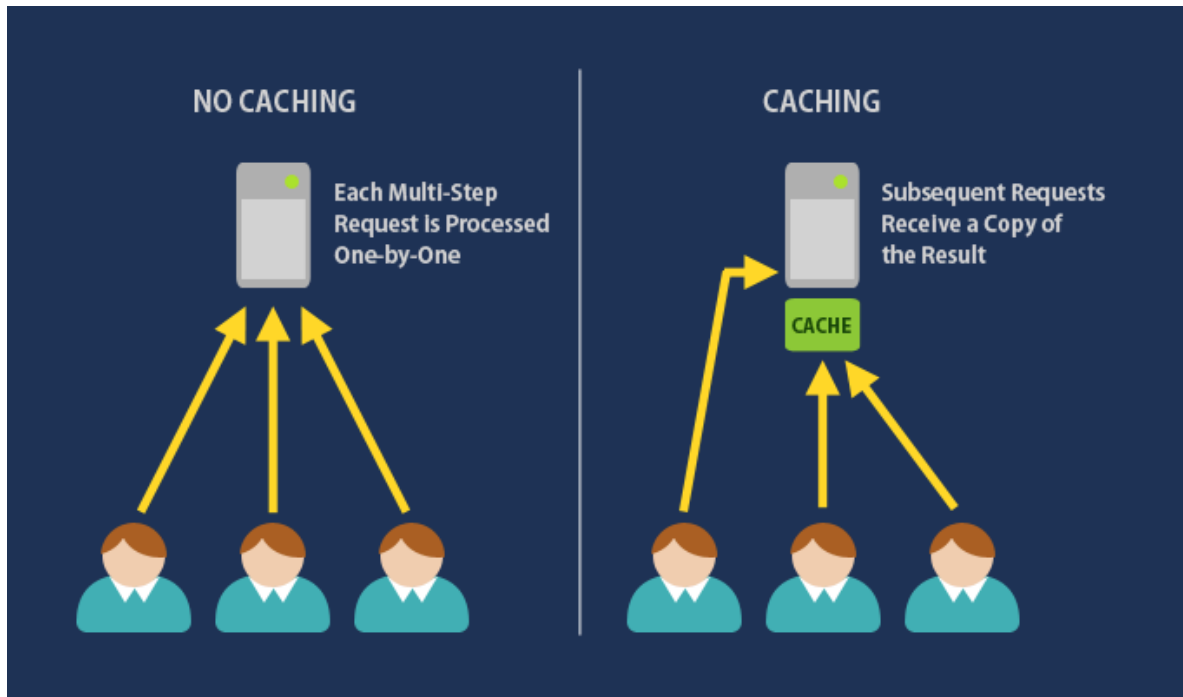
- Chọn tab Application > Storage > Cache Storage.
- Xem danh sách tài nguyên được lưu trữ bởi Service Workers, bao gồm:
  - Tên file
  - Kích thước
  - Thời điểm lưu cache
- Có thể xóa cache thủ công nếu cần kiểm thử lại từ đầu.
- Kiểm tra tình trạng dung lượng bộ nhớ cache nếu phát hiện các lỗi tải chậm.

Browser Cache không chỉ là kỹ thuật tối ưu hiệu năng, mà còn là công cụ thiết yếu trong việc phát triển các hệ thống web tốc độ cao, đặc biệt trong các phần mềm quản lý y tế hiện đại.

## Chương 2. Tổng quan về cache

### 2.1. Cache là gì?

- Bộ nhớ đệm: là phần cứng hoặc phần mềm dùng để lưu trữ tạm thời bản sao của dữ liệu đã được truy cập, tính toán trước đó của máy chủ, ứng dụng hay trình duyệt để nội dung có thể nhanh chóng được gọi lại ra màn hình khi truy cập lại trong quá trình sử dụng. Bộ nhớ đệm còn có thể phát triển theo hướng lưu nội dung vào bộ nhớ hoặc file văn bản trong khoảng thời gian nhất định và mọi người đều có thể truy cập được nội dung trong cache (không chỉ người dùng ban đầu) cho đến khi nó hết hạn.



- Caching, hay kỹ thuật lưu đệm, là quá trình lưu trữ tạm thời các dữ liệu hoặc lệnh thường xuyên truy cập trong bộ nhớ cache. Mục đích của caching là cải thiện hiệu suất hệ thống bằng cách giảm thiểu thời gian truy xuất dữ liệu từ các nguồn tài nguyên chậm hơn như bộ nhớ chính, ổ cứng hoặc server gốc bằng cách lưu trữ các dữ liệu thường xuyên truy cập trong bộ nhớ, tránh việc phải truy cập nhiều lần tới server gốc.
- Caching là một kỹ thuật quản lý dữ liệu nhằm lưu trữ tạm thời các thông tin thường xuyên được truy cập trong một vùng lưu trữ tốc độ cao (cache) để giảm thời gian truy xuất và tăng tốc độ xử lý.

### 2.2. Lịch sử của cache

- Sự hình thành của phương pháp caching đã được lên ý tưởng từ rất lâu về trước đây, bắt đầu từ những năm đầu của máy tính. Ý tưởng về việc lưu trữ tạm thời dữ liệu để tăng tốc độ truy cập được Maurice Wilkes đưa ra năm 1967 khi ông nhận thấy về sự cải thiện đáng kể nếu như có một loại bộ nhớ nhỏ, nhanh hơn để lưu trữ dữ liệu và hướng dẫn CPU truy cập thường xuyên.

- Maurice Wilkes ra đời vào ngày 26/6/1913 tại Dudley, Staffordshire, với cha là Vincent và mẹ là Ellen Wilkes. Hành trình học vấn của Wilkes trải qua những thành tựu đáng nể. Ông theo học tại Trường King Edward VI ở Stourbridge trước khi gia nhập Đại học St John, Cambridge vào năm 1931 và đạt được bằng cử nhân năm 1934. Sau đó, ông tiếp tục nghiên cứu ở Trường Cambridge, và vào năm 1937, ông đã đạt được bằng tiến sĩ sau khi thực hiện nghiên cứu về sự lan truyền của sóng vô tuyến trong tầng điện ly tại Phòng thí nghiệm Cavendish.
- Sau Thế chiến II, ông tham gia vào công việc nghiên cứu về radar trước khi trở lại Cambridge vào năm 1945. Ông đã dẫn dắt Phòng thí nghiệm Toán học từ năm 1946 đến 1970, sau đó nó trở thành Phòng thí nghiệm Máy tính Cambridge, một nơi ông vẫn tiếp tục đóng vai trò quan trọng.
- Năm 1980, Wilkes chuyển hướng sang lĩnh vực công nghệ thông tin, làm việc cho các tổ chức như Tập đoàn Thiết bị Kỹ thuật số ở Massachusetts và Phòng thí nghiệm Nghiên cứu Olivetti và Oracle ở Cambridge, Anh. Ông tiếp tục đóng góp cho ngành máy tính đến những năm cuối đời, được ghi nhận và tưởng nhớ qua nhiều giải thưởng danh giá trong ngành, bao gồm Giải thưởng ACM Turing và Giải thưởng Eckert-Mauchly.
- Maurice Wilkes được biết đến không chỉ là một nhà khoa học hàng đầu mà còn là một nhà lãnh đạo xuất sắc, được tôn vinh bởi nhiều tổ chức uy tín như Hiệp hội Máy tính Anh, Hiệp hội Hoàng gia và Học viện Kỹ thuật Hoàng gia. Ông cũng nhận được sự tôn kính từ các tổ chức quốc tế như Viện Hàn lâm Khoa học và Nghệ thuật Hoa Kỳ. Năm 2000, ông được vinh danh với tước hiệu hiệp sĩ, là một phần của việc công nhận đóng góp to lớn của ông cho lĩnh vực khoa học và công nghệ.
- Bộ nhớ cache đầu tiên được sử dụng trong máy tính CDC 6600 vào năm 1970. Bộ nhớ cache này chỉ có dung lượng 32 kB và được sử dụng để lưu trữ dữ liệu. Ở thời điểm đó, CDC 6600 là một trong những máy tính siêu máy tính đầu tiên, được phát triển bởi Control Data Corporation (CDC). Thiết kế và phát triển của CDC 6600 chủ yếu do Seymour Cray, một trong những nhà thiết kế máy tính nổi tiếng nhất, thực hiện. CDC 6600 đã đánh dấu một bước ngoặt quan trọng trong lịch sử của máy tính, trở thành siêu máy tính nhanh nhất thế giới vào thời điểm đó và mở đường cho các thiết kế siêu máy tính sau này. Việc áp dụng caching vào siêu máy tính này là một bước phát triển lớn đối với công nghệ đương thời.
- Kể từ đó, bộ nhớ cache đã trở thành một thành phần thiết yếu của tất cả các máy tính. Dung lượng và tốc độ của bộ nhớ cache đã tăng lên đáng kể theo thời gian, và ngày nay chúng có thể được tìm thấy ở nhiều cấp độ khác nhau trong hệ thống phân cấp bộ nhớ. Giai đoạn phát triển của caching còn trải qua nhiều giai đoạn, trong đó có thể nói tới việc áp dụng nó vào các máy tính mini (1972), sử dụng trong máy tính cá nhân (1975), trong siêu máy tính (1980) và dần dần, bộ nhớ đệm ngày càng được

nâng cao về cả dung lượng và tốc độ. Điển hình có thể kể đến việc lần đầu tiên bộ nhớ cache ERAM được giới thiệu lần đầu tiên trong máy tính Compaq Presario 7000(2000) và bộ nhớ cache nhúng được giới thiệu lần đầu trong máy tính Intel Core 2 Duo.

- Trong những năm đầu của Caching, kỹ thuật được sử dụng để cache là sử dụng các bộ nhớ đệm nhỏ lưu trữ dữ liệu gần đây truy cập, giúp giảm thời gian truy cập dữ liệu từ bộ nhớ chính(RAM).
- Bước tiếp tới những năm 1970-1980, ta đến với giai đoạn phát triển bộ nhớ Cache và hệ thống lưu trữ. Lúc này, chúng ta đã sản xuất ra các loại bộ nhớ Cache hiện đại hơn, là bộ nhớ Cache L1, giúp tăng khả năng xử lý với kích thước tuy nhỏ nhưng tốc độ cao. Ngoài ra ta còn sử dụng kỹ thuật Disk Cache, tức sử dụng bộ nhớ RAM làm bộ nhớ đệm cho ổ đĩa cứng, giúp giảm thời gian truy cập đến dữ liệu từ ổ cứng.
- Tới những năm 1990 trở đi, sự phát triển của máy tính đã mở rộng và tối ưu hóa Caching hơn. Với sự ra đời của bộ nhớ Cache L2, nằm giữa Cache L1 và RAM, lớn hơn và chậm hơn nhưng cải thiện về mặt hiệu suất. Đối với các trình duyệt web và máy chủ proxy còn xuất hiện thêm Proxy Cache và Browser Cache, là nơi để lưu trữ các bản sao của nội dung web và cache trên trình duyệt để lưu trữ các tệp tin tạm thời, giảm tải mạng và tăng tốc độ truy cập.
- Giai đoạn những năm 2000 là giai đoạn phát triển Caching trong Hệ thống Phân Tán và Cơ sở dữ liệu. Bộ nhớ cache L3 được giới thiệu, chia sẻ giữa các lõi CPU để tăng cường hiệu suất xử lý đa lõi. Distributed Cache là kỹ thuật sử dụng các hệ thống cache phân tán như Memcached và Redis, cho phép lưu trữ và truy xuất dữ liệu từ nhiều node trong hệ thống phân tán. Các ông lớn như Facebook và Twitter là những người sử dụng kỹ thuật này. Query cache trong cơ sở dữ liệu sử dụng bằng cách lưu trữ các kết quả truy vấn từ SQL. Điển hình có thể kể đến 2 database phổ biến sử dụng kỹ thuật này là MySQL và PostgreSQL.
- Những năm 2010 là những năm tối ưu hóa Caching với các hệ thống hiện đại gần với ngày nay. Khi số lượng dữ liệu truy vấn trở nên khổng lồ, bắt đầu có sự xuất hiện của các CDN(Content Delivery Network). CDN sử dụng các mạng phân phối nội dung để lưu trữ bản sao của các hệ thống web tại các máy chủ gần người dùng, giảm độ trễ và tăng tốc độ truy cập. Cloudfare hay Amazon CloudFront là các dịch vụ tiêu biểu mà người dùng có thể dễ biết đến nhất. In-Memory Caching là kỹ thuật đã nói ở trên, sử dụng RAM để cache dữ liệu, được Redis và Memcached tiếp tục phát triển và sử dụng rộng rãi hơn trong các hệ thống. Ngoài ra, với sự phát triển của máy tính, còn xuất hiện thêm một loại kỹ thuật là Edge Computing Cache, hay Cache tại các thiết bị Edge gần người dùng nhất nhằm giảm thiểu tối đa thời gian trễ do khoảng cách địa lý.

- Trở về những năm gần đây, Caching được sử dụng trong các hệ thống AI và Cloud Computing. AI và Machine Learning Caching là kỹ thuật dùng Caching để lưu trữ các mô hình AI và dữ liệu đã được training, giúp giảm thời gian training và dự đoán. Serverless Computing Caching sử dụng các nền tảng serverless để tăng hiệu suất của các hàm không trạng thái. Ngoài ra, việc caching bây giờ có thể được sử dụng trên các kiến trúc hybrid và multi-cloud, giúp giảm tải chi phí lưu trữ hay phân cứng cho những người muốn sử dụng hệ thống lưu trữ.
- Caching đã trải qua một quá trình phát triển dài và không ngừng cải tiến. Từ những bộ nhớ đệm cơ bản trong các hệ thống máy tính đầu tiên, caching đã mở rộng và trở nên phức tạp hơn với sự ra đời của các kỹ thuật và công nghệ mới. Caching hiện nay được sử dụng rộng rãi trong nhiều lĩnh vực, từ CPU và hệ thống lưu trữ đến mạng, cơ sở dữ liệu, và các ứng dụng phân tán hiện đại, góp phần quan trọng vào việc tối ưu hóa hiệu suất và trải nghiệm người dùng.
- Caching đã trải qua một quá trình phát triển dài và không ngừng cải tiến. Từ những bộ nhớ đệm cơ bản trong các hệ thống máy tính đầu tiên, caching đã mở rộng và trở nên phức tạp hơn với sự ra đời của các kỹ thuật và công nghệ mới. Caching hiện nay được sử dụng rộng rãi trong nhiều lĩnh vực, từ CPU và hệ thống lưu trữ đến mạng, cơ sở dữ liệu, và các ứng dụng phân tán hiện đại, góp phần quan trọng vào việc tối ưu hóa hiệu suất và trải nghiệm người dùng.
- Tóm tắt lịch sử của cache qua các năm
  - 1967: Maurice Wilkes đề xuất ý tưởng về bộ nhớ cache.
  - 1970: Máy tính CDC 6600 sử dụng bộ nhớ cache lần đầu tiên.
  - 1972: Bộ nhớ cache được sử dụng trong máy tính mini.
  - 1975: Bộ nhớ cache được sử dụng trong máy tính cá nhân.
  - 1980: Bộ nhớ cache được sử dụng trong máy tính siêu máy tính.
  - 1990: Bộ nhớ cache trở thành một thành phần tiêu chuẩn trong tất cả các máy tính.
  - 2000: Bộ nhớ cache trở nên lớn hơn và nhanh hơn

## 2.3. Ưu điểm và nhược điểm của cache

### 2.3.1. Ưu điểm của Cache

#### Tăng tốc độ truy xuất dữ liệu

- Giảm thời gian phản hồi: Khi một hệ thống lưu dữ liệu trong cache, các lần truy cập tiếp theo có thể lấy dữ liệu từ cache thay vì truy xuất từ nguồn gốc chậm hơn (như cơ sở dữ liệu hoặc máy chủ từ xa). Điều này giảm thời gian truy vấn, đặc biệt hữu ích trong các ứng dụng yêu cầu phản hồi nhanh, như ứng dụng web và di động.



- Truy xuất từ bộ nhớ nhanh hơn: Cache thường sử dụng các công nghệ lưu trữ nhanh hơn như RAM, thay vì sử dụng các thiết bị lưu trữ chậm hơn như ổ cứng hoặc mạng, giúp hệ thống đáp ứng các yêu cầu nhanh chóng.

*Ví dụ:* Trong trình duyệt web, các tệp như CSS, JavaScript và hình ảnh của một trang web được lưu trong cache. Khi người dùng truy cập lại trang đó, trình duyệt chỉ cần tải lại những tệp này từ cache thay vì từ máy chủ, giúp trang web tải nhanh hơn.

### **Giảm tải cho hệ thống gốc**

- Giảm truy vấn đến cơ sở dữ liệu: Thay vì thực hiện các truy vấn tốn tài nguyên và thời gian, cache lưu lại kết quả của các truy vấn thường xuyên được yêu cầu. Điều này giúp giảm số lượng truy vấn đến cơ sở dữ liệu và giảm thiểu tình trạng quá tải.
- Giảm băng thông và tải máy chủ: Bằng cách phục vụ các yêu cầu từ cache thay vì phải chuyển dữ liệu từ các nguồn xa (như máy chủ gốc), hệ thống có thể giảm sử dụng băng thông mạng, giúp các máy chủ gốc và hạ tầng mạng không bị quá tải.

*Ví dụ:* Các hệ thống như CDN (Content Delivery Network) lưu trữ các bản sao nội dung web trên các máy chủ gần người dùng hơn, giảm số lượng yêu cầu đến máy chủ gốc và tiết kiệm băng thông.

### **Cải thiện trải nghiệm người dùng**

- Thời gian phản hồi nhanh hơn: Thời gian tải trang hoặc ứng dụng nhanh hơn giúp người dùng không phải chờ đợi, mang lại trải nghiệm mượt mà và thoải mái hơn.
- Tăng tính khả dụng: Khi hệ thống gặp sự cố hoặc có sự cố kết nối với nguồn dữ liệu gốc, các dữ liệu trong cache vẫn có thể được phục vụ cho người dùng, giúp hệ thống có khả năng tiếp tục hoạt động trong một số trường hợp.

*Ví dụ:* Trong các ứng dụng di động, dữ liệu có thể được lưu trong cache để người dùng có thể truy cập ngay cả khi kết nối mạng yếu hoặc tạm thời bị mất.

### **Tiết kiệm tài nguyên hệ thống**

- Giảm chi phí tính toán và truy xuất dữ liệu: Khi sử dụng cache, các tác vụ tính toán phức tạp hoặc truy vấn tốn thời gian chỉ cần thực hiện một lần, sau đó kết quả được lưu trữ và tái sử dụng. Điều này giúp hệ thống không cần phải lặp lại các tác vụ đòi hỏi nhiều tài nguyên mỗi khi có yêu cầu tương tự.
- Tối ưu hóa hiệu suất bộ xử lý (CPU): Các hệ thống máy tính và vi xử lý sử dụng cache để lưu trữ các lệnh hoặc dữ liệu được sử dụng thường xuyên, giảm thời gian CPU phải đợi dữ liệu từ bộ nhớ chính (RAM hoặc ổ đĩa), tăng hiệu suất xử lý tổng thể.

*Ví dụ:* Bộ nhớ cache của CPU giúp giảm độ trễ giữa CPU và bộ nhớ chính (RAM), từ đó tăng tốc độ xử lý các tác vụ mà CPU cần thực hiện.

### **Tăng khả năng mở rộng**

- Hỗ trợ xử lý tải cao: Khi lưu lượng truy cập tăng đột biến, cache giúp giảm tải cho các hệ thống chính bằng cách phục vụ nhiều yêu cầu từ cache. Điều này cho phép hệ thống xử lý được nhiều người dùng hoặc yêu cầu cùng lúc mà không làm giảm hiệu suất.
- Phân phối nội dung hiệu quả: Trong các hệ thống phân tán hoặc ứng dụng toàn cầu, cache (đặc biệt là CDN) giúp giảm khoảng cách giữa người dùng và nguồn dữ liệu, cải thiện hiệu quả phân phối nội dung mà không cần phải mở rộng hạ tầng máy chủ chính.

*Ví dụ:* Một trang web thương mại điện tử có thể sử dụng cache để giảm tải cho cơ sở dữ liệu và máy chủ ứng dụng trong các sự kiện lớn như ngày mua sắm trực tuyến, nơi số lượng truy cập có thể tăng lên hàng nghìn lần.

### **Tăng tính ổn định**

- Bảo vệ hệ thống khỏi sự cố quá tải: Cache giúp hệ thống tránh được tình trạng quá tải trong các giai đoạn lưu lượng truy cập đột biến, giúp hệ thống duy trì ổn định. Điều này đặc biệt quan trọng với các hệ thống có quy mô lớn hoặc các ứng dụng mà trải nghiệm người dùng là yếu tố then chốt.
- Giảm tần suất truy cập vào các tài nguyên đắt đỏ: Các tài nguyên như cơ sở dữ liệu phức tạp, máy chủ từ xa hoặc dịch vụ ngoài thường đắt đỏ về chi phí và thời gian truy cập. Cache giảm thiểu số lần truy cập trực tiếp vào các tài nguyên này, bảo vệ hệ thống khỏi các cuộc tấn công DDoS (Distributed Denial of Service) hoặc tải không mong muốn.

*Ví dụ:* Các trang web lớn sử dụng hệ thống cache phức tạp để đảm bảo rằng khi có lượng người dùng lớn truy cập đồng thời, trang web vẫn có thể đáp ứng một cách ổn định và nhanh chóng.

### **Tối ưu hóa cho hệ thống phân tán**

- Giảm độ trễ giữa các khu vực địa lý khác nhau: Cache giúp dữ liệu được lưu trữ gần người dùng hơn, đặc biệt trong các hệ thống phân tán toàn cầu. Điều này giúp giảm độ trễ khi truyền dữ liệu giữa các khu vực địa lý xa nhau, cải thiện hiệu suất cho người dùng ở mọi nơi.
- Tăng hiệu quả trong hệ thống phân tán: Trong các hệ thống nhiều máy chủ, cache được sử dụng để lưu trữ dữ liệu chung hoặc cục bộ nhằm giảm thời gian truy xuất giữa các node, tăng hiệu quả hoạt động chung của toàn hệ thống.

*Ví dụ:* CDN giúp các trang web và dịch vụ trực tuyến lưu trữ các bản sao dữ liệu trên các máy chủ phân phối tại nhiều nơi, giảm thiểu độ trễ và cải thiện hiệu suất khi người dùng từ các khu vực khác nhau truy cập.

### **Linh hoạt trong quản lý dữ liệu**

- Hỗ trợ nhiều loại dữ liệu: Cache có thể lưu trữ đa dạng các loại dữ liệu, từ kết quả truy vấn cơ sở dữ liệu, nội dung web, đến các tệp lớn như hình ảnh, video. Điều này giúp tăng cường hiệu quả cho các hệ thống xử lý nhiều loại dữ liệu khác nhau.
- Cấu hình và điều chỉnh dễ dàng: Cache có thể được cấu hình với các chiến lược khác nhau (như LRU, FIFO) hoặc tùy chỉnh TTL theo loại dữ liệu hoặc yêu cầu ứng dụng. Điều này giúp các nhà phát triển linh hoạt hơn trong việc quản lý và tối ưu hóa hiệu suất.

*Ví dụ:* Trong một ứng dụng web, nhà phát triển có thể tùy chỉnh TTL cho từng loại tài nguyên, ví dụ như lưu trữ hình ảnh trong cache lâu hơn so với các dữ liệu động như kết quả truy vấn cơ sở dữ liệu.

### 2.3.2. Nhược điểm của Cache

#### Dữ liệu lỗi thời

- Dữ liệu không cập nhật: Một trong những thách thức lớn của cache là lưu trữ dữ liệu có thể trở nên lỗi thời so với nguồn dữ liệu gốc. Nếu dữ liệu trong cache không được làm mới kịp thời khi có thay đổi tại nguồn gốc, hệ thống có thể trả về các kết quả sai hoặc không chính xác.
- Cache Invalidation (Xóa bỏ cache cũ): Việc quản lý và cập nhật cache khi dữ liệu gốc thay đổi là một quá trình phức tạp. Nếu không có cơ chế cache invalidation hiệu quả, dữ liệu trong cache có thể không còn phù hợp, dẫn đến người dùng truy cập thông tin không chính xác.

*Ví dụ:* Trong một ứng dụng thương mại điện tử, nếu giá sản phẩm thay đổi nhưng cache không được làm mới kịp thời, người dùng có thể thấy giá cũ thay vì giá mới, gây ra trải nghiệm không tốt hoặc thậm chí sai lệch trong giao dịch.

#### Quản lý dung lượng Cache

- Giới hạn dung lượng: Cache có dung lượng giới hạn, đặc biệt là khi sử dụng bộ nhớ RAM. Khi cache đầy, hệ thống phải loại bỏ các mục cũ để nhường chỗ cho dữ liệu mới. Nếu không có chiến lược quản lý tốt, việc loại bỏ sai dữ liệu có thể làm giảm hiệu quả của cache.
- Chi phí lưu trữ: Để lưu trữ dữ liệu trong cache, đặc biệt là với các hệ thống lớn hoặc dữ liệu có dung lượng lớn (ví dụ: hình ảnh, video), có thể tốn kém tài nguyên bộ nhớ hoặc dung lượng ổ đĩa, dẫn đến chi phí tăng lên nếu không được quản lý hợp lý.

*Ví dụ:* Các ứng dụng lưu trữ dữ liệu lớn (như video stream) có thể gặp vấn đề khi cache quá tải và không đủ bộ nhớ để lưu trữ nhiều dữ liệu cùng lúc.

#### Cache miss có thể xảy ra thường xuyên

- Hiệu suất bị ảnh hưởng: Khi hệ thống gặp cache miss (khi dữ liệu không có trong cache), nó phải truy cập vào nguồn dữ liệu gốc, làm tăng thời gian phản hồi. Nếu

cache miss xảy ra thường xuyên, hiệu suất hệ thống có thể bị ảnh hưởng nghiêm trọng.

- Cold Cache (Cache lạnh): Sau khi khởi động lại hệ thống hoặc khi cache mới được thiết lập, cache chưa có dữ liệu, dẫn đến tình trạng cache miss nhiều hơn và hiệu suất ban đầu thấp hơn. Quá trình làm đầy cache để đạt hiệu quả tốt nhất có thể mất thời gian.

*Ví dụ:* Trong một trang web lớn, khi người dùng truy cập vào lần đầu sau khi cache đã bị xóa, trang sẽ tải chậm hơn vì hệ thống phải lấy dữ liệu từ nguồn gốc thay vì từ cache.

### **Chi phí đồng bộ và quản lý**

- Cache Consistency (Đồng bộ cache): Trong các hệ thống phân tán hoặc các ứng dụng có nhiều điểm lưu trữ cache, việc đảm bảo tất cả các cache đều nhất quán với dữ liệu gốc là một thách thức lớn. Dữ liệu có thể thay đổi ở một nơi, nhưng các bản sao trong cache ở các nơi khác không được cập nhật kịp thời, dẫn đến sự không đồng bộ.
- Chi phí và phức tạp: Để đảm bảo tính nhất quán của cache với dữ liệu gốc, cần có các cơ chế quản lý phức tạp như cache invalidation, cache refresh, hoặc các chiến lược đồng bộ dữ liệu. Điều này có thể làm tăng chi phí phát triển và bảo trì hệ thống.

*Ví dụ:* Một hệ thống ứng dụng phân tán với nhiều máy chủ có thể lưu trữ bản sao dữ liệu trong cache riêng biệt. Nếu một máy chủ cập nhật dữ liệu, các máy chủ khác có thể vẫn còn dữ liệu cũ trong cache, dẫn đến tình trạng không đồng bộ.

### **Chi phí khởi tạo và làm đầy Cache**

- Thời gian khởi tạo lâu: Cache cần thời gian để làm đầy và hoạt động hiệu quả, đặc biệt là khi dữ liệu phải được tải từ nguồn gốc. Trong giai đoạn đầu, hệ thống sẽ trải qua nhiều cache miss và hiệu suất sẽ không được tối ưu.
- Chi phí xử lý cao: Khi cache không có dữ liệu, việc lấy dữ liệu từ nguồn gốc có thể tốn nhiều tài nguyên xử lý, đặc biệt với các hệ thống có dữ liệu lớn hoặc phức tạp. Điều này làm tăng tải lên các hệ thống nguồn, giảm hiệu suất tổng thể trong quá trình khởi tạo.

*Ví dụ:* Trong một ứng dụng web, sau khi khởi động lại server, cache có thể cần một khoảng thời gian để lưu lại các tài nguyên thường xuyên được truy cập. Trong thời gian này, trang web có thể tải chậm hơn cho người dùng.

### **Tăng độ phức tạp trong kiến trúc hệ thống**

- Phức tạp hơn trong quản lý hệ thống: Việc triển khai và duy trì hệ thống cache đòi hỏi cấu trúc phức tạp hơn và cần phải có các chiến lược quản lý cache hiệu quả.

Điều này có thể làm tăng độ phức tạp của hệ thống và yêu cầu thêm kiến thức chuyên môn để xử lý các vấn đề phát sinh liên quan đến cache.

- Khả năng gặp lỗi cao hơn: Hệ thống cache thêm một lớp trung gian giữa ứng dụng và nguồn dữ liệu gốc, làm tăng nguy cơ gặp các lỗi kỹ thuật như cache corruption (dữ liệu cache bị hỏng), cache miss nhiều lần, hoặc lỗi đồng bộ dữ liệu.

*Ví dụ:* Một hệ thống lớn với nhiều tầng cache khác nhau (như cache trong bộ nhớ, cache cơ sở dữ liệu, và cache mạng) sẽ phức tạp hơn để quản lý và có thể dẫn đến các lỗi khi dữ liệu không được đồng bộ hóa đúng cách.

### **Bảo mật và quyền riêng tư**

- Dữ liệu nhạy cảm trong cache: Cache có thể lưu trữ các thông tin nhạy cảm như thông tin cá nhân, thông tin đăng nhập, hoặc dữ liệu nhạy cảm khác. Nếu cache không được bảo mật tốt, tin tặc có thể truy cập vào các thông tin nhạy cảm này một cách dễ dàng.
- Rủi ro bị tấn công: Cache có thể là mục tiêu của các cuộc tấn công, đặc biệt là các cuộc tấn công kiểu cache poisoning (tấn công làm nhiễm độc cache), trong đó tin tặc đưa dữ liệu độc hại vào cache và từ đó làm hỏng hoặc thao túng dữ liệu mà người dùng hoặc hệ thống truy cập.

*Ví dụ:* Trong một hệ thống web, nếu dữ liệu nhạy cảm như thông tin thanh toán được lưu trữ trong cache không được mã hóa, các cuộc tấn công có thể dễ dàng truy cập và khai thác thông tin đó.

### **Hiệu suất không đồng đều**

- Không đảm bảo hiệu suất liên tục: Hiệu quả của cache phụ thuộc rất nhiều vào tỷ lệ cache hit (tức là số lần dữ liệu được tìm thấy trong cache). Nếu tỷ lệ này thấp, hiệu suất tổng thể của hệ thống sẽ giảm sút, do hệ thống phải thường xuyên truy cập vào nguồn dữ liệu gốc.
- Cache không phù hợp cho mọi loại dữ liệu: Không phải tất cả dữ liệu đều có thể được cache hiệu quả. Các dữ liệu thay đổi liên tục hoặc các dữ liệu ít được truy cập không mang lại lợi ích lớn khi lưu trữ trong cache.

*Ví dụ:* Trong một hệ thống web, nếu trang web có nội dung động thay đổi liên tục (như tin tức, dữ liệu thời gian thực), cache có thể không cải thiện hiệu suất đáng kể vì dữ liệu sẽ nhanh chóng trở nên lỗi thời.

## **2.4. Các thông số và yếu tố hỗ trợ quan trọng thường sử dụng trong việc cache**

- Tỷ lệ trúng cache (Cache Hit Ratio): Đây là tỷ lệ phần trăm của các yêu cầu dữ liệu mà hệ thống cache có thể phục vụ trực tiếp từ cache mà không cần truy cập tới

nguồn dữ liệu gốc. Tỷ lệ trúng cache cao thường là một chỉ số cho thấy hiệu suất tốt của hệ thống cache.

- Tỷ lệ trượt cache (Cache Miss Ratio): Đây là tỷ lệ phần trăm của các yêu cầu dữ liệu không được phục vụ từ cache mà cần truy cập đến nguồn dữ liệu gốc. Tỷ lệ này càng thấp càng cho thấy hiệu quả của hệ thống cache. Tỷ lệ trượt =  $1 - \text{Cache Hit Ratio}$ .
- Thời gian sống (Time To Live - TTL): Đây là khoảng thời gian mà một mục dữ liệu được phép tồn tại trong cache trước khi nó được coi là hết hạn và bị loại bỏ. TTL giúp đảm bảo rằng dữ liệu trong cache luôn cập nhật và tránh lưu trữ các thông tin cũ lỗi thời. TTL ngắn giúp dữ liệu tươi mới, TTL dài giúp giảm tần suất truy vấn nguồn gốc.
- Thông lượng (Throughput): Đây là số lượng yêu cầu mà hệ thống cache có thể xử lý trong một đơn vị thời gian (ví dụ: yêu cầu/giây). Thông lượng cao phản ánh khả năng xử lý hiệu quả của hệ thống cache trong môi trường tải cao.
- Độ trễ truy cập (Access Latency): Là thời gian cần thiết để truy xuất một mục dữ liệu từ cache. Độ trễ thấp đồng nghĩa với việc dữ liệu được phục vụ nhanh chóng, cải thiện hiệu suất tổng thể.
- Kích thước cache (Cache Size): Dung lượng bộ nhớ được phân bổ cho hệ thống cache. Kích thước cache quá nhỏ có thể làm giảm tỷ lệ cache hit, trong khi kích thước quá lớn có thể gây lãng phí tài nguyên.
- Thời gian truy cập trung bình (Average Access Time): Đây là thời gian trung bình mà một yêu cầu dữ liệu phải mất để được phục vụ từ cache. Thời gian truy cập thấp là một chỉ số cho thấy hiệu suất tốt của cache.
- Chính sách loại bỏ (Eviction Policy): Là cách mà hệ thống quyết định loại bỏ dữ liệu cũ khi cache đầy. Các chính sách phổ biến gồm LRU (Least Recently Used), LFU (Least Frequently Used), FIFO (First In First Out),... Việc chọn chính sách phù hợp ảnh hưởng lớn đến hiệu suất cache.
- Tính sẵn sàng (Availability): Là khả năng cache tiếp tục phục vụ yêu cầu ngay cả khi xảy ra sự cố. Các cơ chế như cache replication (sao chép) và failover (chuyển đổi khi lỗi) thường được dùng để đảm bảo tính sẵn sàng.
- Khả năng mở rộng (Scalability): Khả năng hệ thống cache xử lý hiệu quả khi lưu lượng truy cập hoặc kích thước dữ liệu tăng lên. Một hệ thống cache tốt cần dễ dàng mở rộng mà không làm giảm hiệu suất.
- Quản lý bộ nhớ (Memory Management): Các hệ thống cache phải có cơ chế quản lý bộ nhớ hiệu quả để đảm bảo sử dụng tài nguyên bộ nhớ một cách hiệu quả nhất. Điều này bao gồm cơ chế như cache eviction (loại bỏ dữ liệu không cần thiết), caching policies (chính sách lưu trữ dữ liệu), và phân phối tài nguyên bộ nhớ.
- Bảo mật (Security): Bảo mật dữ liệu trong cache là một vấn đề quan trọng, đặc biệt khi dữ liệu nhạy cảm được lưu trữ trong cache. Các chuẩn mực bảo mật có thể bao gồm mã hóa dữ liệu, kiểm tra danh tính, và giới hạn quyền truy cập vào cache.

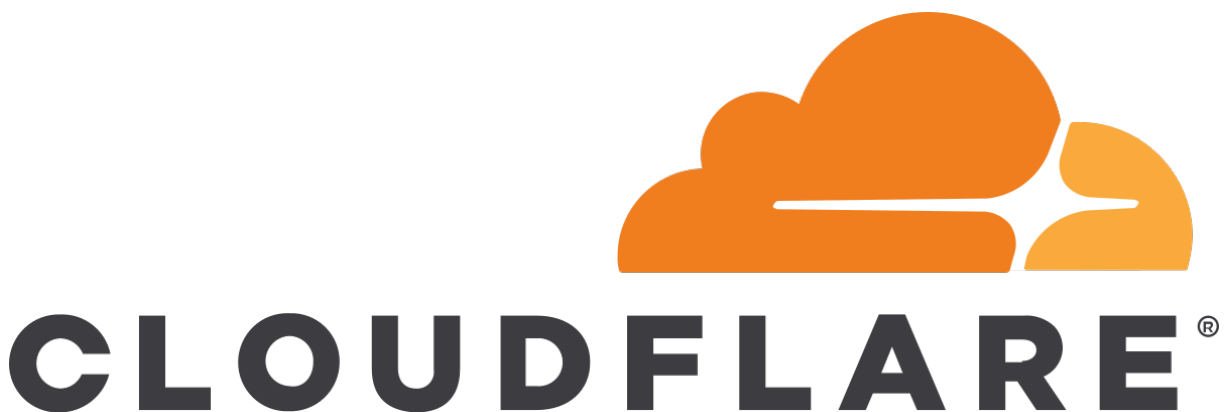
- Sao lưu và khôi phục (Backup and Recovery): Đảm bảo rằng dữ liệu trong cache có thể được sao lưu và khôi phục một cách an toàn là một phần quan trọng của chuẩn mực của cache, đặc biệt là trong các môi trường sản xuất.
- Kiểm thử và đánh giá hiệu suất (Testing and Performance Evaluation): Các hệ thống cache cần được kiểm thử kỹ lưỡng và đánh giá hiệu suất để đảm bảo rằng chúng đáp ứng được các yêu cầu hiệu suất và độ tin cậy.

## 2.5. Cache invalidation

- Cache invalidation (vô hiệu hóa bộ đệm) là quá trình cập nhật hoặc loại bỏ các mục trong bộ nhớ cache khi dữ liệu gốc bị thay đổi, nhằm đảm bảo dữ liệu trong cache luôn đồng bộ chính xác với nguồn dữ liệu chính, nâng cao tính nhất quán của cache (cache consistency) và ngăn ngừa lỗi.
- Vậy vấn đề đặt ra là nếu không thực hiện cache invalidation kịp thời, hệ thống có thể trả về dữ liệu cũ (stale), gây sai lệch hoặc lỗi cho người dùng.
- Cache invalidation giải quyết một trong những thách thức lớn nhất trong hệ thống phân tán: đảm bảo tính nhất quán giữa dữ liệu được lưu trong cache và dữ liệu trong cơ sở dữ liệu gốc. Khi một hệ thống sử dụng caching để tối ưu hiệu suất, nó phải đối mặt với vấn đề "cache coherence" - đảm bảo rằng các thay đổi dữ liệu được phản ánh chính xác trong tất cả các bản sao cache trên toàn hệ thống. Về bản chất, cache invalidation giải quyết câu hỏi: **"Khi nào và làm thế nào để cập nhật hoặc loại bỏ dữ liệu cache không còn chính xác?"**

*Đó cũng là lý do cache invalidation được xem là một trong hai vấn đề khó nhất trong ngành khoa học máy tính*

### 2.5.1. Cloudflare: công nghệ và công cụ thực tiễn



Cloudflare cung cấp một CDN toàn cầu với nhiều công cụ để làm mất hiệu lực cache linh hoạt.

#### 2.5.1.1. Purge cache (xóa cache thủ công)

- Cloudflare cho phép xóa cache theo nhiều cách. Thông thường, Purge by URL (single-file purge) được khuyến dùng để xóa từng tệp. Ngoài ra, còn có Purge by

Hostname/Prefix (xóa theo miền hoặc theo tiền tố URL) và Purge Everything (xóa toàn bộ cache của domain). Tất cả các gói (Free/Pro/Business/Enterprise) đều hỗ trợ xóa theo URL, hostname, tag, prefix và purge everything. Nhờ hệ thống “Instant Purge” mới, Cloudflare có thể loại bỏ nội dung cache toàn cầu trong vòng <150ms (P50) khi purge bằng tags, hostname hoặc prefix.

- Ưu điểm: cập nhật tức thì, người dùng sẽ nhận nội dung mới ngay lập tức.
- Nhược điểm: nếu xóa nhiều tệp cùng lúc, CDN và origin có thể bị tải lại mạnh; tuy nhiên Cloudflare đã tối ưu hạ tầng để hạn chế độ trễ và kết xuất nhanh.

**Lưu ý:** thao tác purge không ảnh hưởng đến bộ nhớ đệm của trình duyệt, chỉ áp dụng trên các edge nodes của Cloudflare

#### **2.5.1.2. Cache-Tag (cache nhãn)**

- Bằng cách thêm header Cache-Tag với một hoặc nhiều nhãn trên phản hồi từ origin, mỗi tài nguyên được đánh dấu với tag riêng. Khi cần invalidation, sử dụng API hoặc dashboard để purge theo tag đó, tức xóa đồng loạt tất cả nội dung mang tag tương ứng.

*Ví dụ:* tag user-123 trên tất cả API liên quan tới người dùng

123. Ưu điểm: xóa đồng loạt dễ dàng mà không cần liệt kê từng URL. Nhược

điểm: cần cấu hình server để gắn tag (chỉ hỗ trợ ASCII, độ dài tối đa 1024 ký tự cho tag khi purge) và giới hạn tổng header, tăng công việc phát triển.

#### **2.5.1.3. Edge Cache TTL**

- Cloudflare cho phép thiết lập Edge Cache TTL (bằng Cache Rules hoặc Dashboard) để kiểm soát thời gian tối đa lưu trên edge servers

*Ví dụ:* Free Plan có TTL tối thiểu 2 giờ, Pro/Biz có thể đặt ngắn hơn (có thể 1s với Enterprise). Nếu origin trả về header max-age dài hơn, Edge TTL cao nhất cũng sẽ chi phối (ít nhất là giá trị thiết lập). Bằng cách này, ta có thể buộc cache tự hết hạn sau một thời gian ngắn tùy ý.

#### **2.5.1.4. Browser Cache TTL**

- Ngoài ra, có thể thiết lập Browser Cache TTL để ghi đè header gốc với trình duyệt người dùng.

*Ví dụ:* đặt cao để cho browser lưu lâu. Lưu ý rằng nếu thay đổi tài nguyên nhưng client còn cache cũ, họ có thể không nhận được ngay; đồng thời, việc purge trên Cloudflare không xóa cache đã lưu trên trình duyệt.



### 2.5.1.5. Tái xác thực (Revalidation)

- Khi tài nguyên trên edge đã hết hạn (hết TTL), Cloudflare tự động gửi conditional request (If-Modified-Since/ETag) tới origin để kiểm tra. Nếu nội dung không đổi, Cloudflare gia hạn TTL mà không tải lại hoàn toàn.
- *Đặc biệt*, khi có nhiều yêu cầu đồng thời cho tài nguyên hết hạn, Cloudflare sử dụng cơ chế khoá cache để chỉ một request được gửi tới origin, các request còn lại sẽ được trả tạm dữ liệu cũ kèm trạng thái UPDATING trong khi đang chờ lấy nội dung mới. Tức là Cloudflare triển khai stale-while-revalidate mặc định: 999 request tiếp theo sử dụng nội dung cũ trong lúc một request duyệt ra origin.
- Điều này giúp giảm tải cho origin và vẫn đảm bảo cuối cùng cache được cập nhật mới.

*Tóm lại, Cloudflare hỗ trợ nhiều kỹ thuật invalidation: từ purge tức thì (URL/tag/prefix/đại lý), đến tự hết hạn (TTL) và xác thực lại (ETag/Last-Modified). Việc lựa chọn phù hợp phụ thuộc vào nhu cầu (ví dụ dữ liệu thay đổi nhiều thì purge/tag gần nhãn, nếu thay đổi ít thì có thể dùng TTL dài + stale-while-revalidate). Với hạ tầng hiện tại, Cloudflare cam kết khả năng purge nhanh chóng toàn cầu, giúp rủi ro phục vụ nội dung cũ được giảm tối đa.*

## 2.6. Cache replacement policies

- Trong lĩnh vực điện toán, các chính sách thay thế bộ nhớ đệm (còn gọi là thuật toán thay thế bộ nhớ đệm hoặc thuật toán bộ đệm) là những hướng dẫn hoặc thuật toán tối ưu hóa mà một chương trình máy tính hoặc một cấu trúc được duy trì bởi phần cứng có thể sử dụng để quản lý một bộ nhớ đệm thông tin. Việc sử dụng bộ nhớ đệm giúp cải thiện hiệu suất bằng cách giữ lại các mục dữ liệu gần đây hoặc thường xuyên được sử dụng trong các vị trí bộ nhớ nhanh hơn, hoặc rẻ hơn về mặt tính toán, so với các kho lưu trữ bộ nhớ thông thường. Khi bộ nhớ đệm đầy, thuật toán phải chọn ra các mục nào sẽ bị loại bỏ để nhường chỗ cho dữ liệu mới.
- Thời gian tham chiếu bộ nhớ trung bình:

$$T = m \times T_m + T_h + E$$

Trong đó:

- $m$  = miss ratio =  $1 - \text{hit ratio}$
- $T_m$  = thời gian truy cập bộ nhớ chính khi có cache miss (hoặc, nếu có cache nhiều tầng, là thời gian truy cập trung bình đến cache tầng kế tiếp)
- $T_h$  = độ trễ: thời gian truy cập cache (giống nhau cho cả hit và miss)
- $E$  = các hiệu ứng phụ, ví dụ như hiệu ứng hàng đợi trong hệ thống đa xử lý

Một bộ nhớ đệm (cache) có hai chỉ số đánh giá chính: độ trễ (latency) và tỉ lệ trúng (hit ratio). Ngoài ra, còn có một số yếu tố phụ khác cũng ảnh hưởng đến hiệu suất của cache.

- Tỷ lệ trúng (hit ratio) mô tả mức độ thường xuyên mà một mục được tìm kiếm thực sự được tìm thấy trong cache.
- Các chính sách thay thế (replacement policies) hiệu quả hơn sẽ theo dõi thêm thông tin về việc sử dụng để cải thiện tỷ lệ trúng đối với một kích thước cache nhất định.
- Độ trễ (latency) mô tả khoảng thời gian từ khi yêu cầu một mục cho đến khi cache có thể trả lại mục đó trong trường hợp trúng (hit). - Các chiến lược thay thế nhanh hơn thường theo dõi ít thông tin sử dụng hơn — hoặc, trong trường hợp cache ánh xạ trực tiếp (direct-mapped cache), không theo dõi gì cả — nhằm giảm thời gian cần để cập nhật thông tin. Mỗi chiến lược thay thế là một sự đánh đổi giữa tỷ lệ trúng và độ trễ.

Việc đo tỷ lệ trúng thường được thực hiện trên các ứng dụng chuẩn (benchmark), và tỷ lệ này thay đổi tùy theo ứng dụng. Các ứng dụng phát video và âm thanh thường có tỷ lệ trúng gần bằng không, bởi vì mỗi bit dữ liệu trong luồng chỉ được đọc một lần (miss bắt buộc), sử dụng xong là không được đọc hoặc ghi lại nữa. Nhiều thuật toán cache (đặc biệt là LRU) cho phép dữ liệu luồng làm đầy cache, đẩy ra những thông tin sẽ sớm được sử dụng lại (gây ô nhiễm cache – cache pollution).

Các yếu tố khác có thể bao gồm kích thước, thời gian truy cập và thời hạn hết hạn của dữ liệu. Tùy thuộc vào kích thước cache, có thể không cần thuật toán loại bỏ nào thêm. Các thuật toán cũng cần duy trì tính nhất quán của cache (cache coherence) khi có nhiều cache cùng sử dụng chung dữ liệu, chẳng hạn như nhiều máy chủ cơ sở dữ liệu cùng cập nhật một tệp dữ liệu chia sẻ.

## **2.6.1. Các chính sách nổi bật**

### **2.6.1.0.1. Thuật toán FIFO (First-In, First-Out)**

Là một trong những giải thuật thay thế trang/bộ nhớ đơn giản và cổ điển nhất, được đặt theo nguyên lý hoạt động của hàng đợi: phần tử nào vào trước thì sẽ được lấy ra trước. Khi áp dụng FIFO cho bộ nhớ đệm, các tài nguyên được nạp vào cache sẽ bị loại bỏ theo thứ tự thời gian mà chúng được thêm vào, bất kể mức độ thường xuyên sử dụng hoặc thời điểm sử dụng gần nhất của chúng.

#### **Ưu điểm:**

- Cài đặt đơn giản: FIFO rất dễ triển khai. Chỉ cần sử dụng một cấu trúc hàng đợi (queue) cơ bản để quản lý thứ tự vào của các phần tử trong cache. Không cần phải tính toán phức tạp về thời gian truy cập hay số lần truy cập.
- Chi phí thấp (low overhead): Do không cần theo dõi thông tin chi tiết về tần suất hoặc thời gian truy cập của từng phần tử, FIFO tiêu tốn ít bộ nhớ và tài nguyên xử lý hơn so với các thuật toán như LRU hay LFU.

- Công bằng về thời gian lưu trữ: Tất cả các phần tử được giữ trong cache một khoảng thời gian gần bằng nhau, giúp đảm bảo tính công bằng theo thời gian mà không phân biệt mức độ "quan trọng" của dữ liệu.

#### **Nhược điểm:**

- Không xem xét frequency hoặc recency: FIFO không quan tâm đến việc phần tử có được truy cập thường xuyên hay gần đây hay không. Do đó, có thể xảy ra trường hợp loại bỏ một phần tử vẫn đang được sử dụng nhiều, chỉ vì nó đã vào cache trước.
- Dễ loại bỏ dữ liệu quan trọng: Trong các hệ thống có tính chất temporal locality (dữ liệu được sử dụng gần nhau về thời gian có khả năng sẽ được sử dụng lại), FIFO hoạt động kém hiệu quả vì nó có thể loại bỏ dữ liệu quan trọng vẫn còn được truy cập thường xuyên.
- Hiệu suất thấp trong nhiều tình huống thực tế: Đặc biệt trong các ứng dụng nơi dữ liệu "nóng" (hot data) cần được giữ lại trong cache lâu hơn, FIFO thường không đáp ứng được hiệu suất như mong đợi. Nó có thể dẫn đến tỷ lệ cache miss cao, ảnh hưởng đến hiệu suất tổng thể của hệ thống.

#### **2.6.1.1. Thuật toán LRU (Least Recently Used)**

Thuật toán LRU loại bỏ phần tử ít được sử dụng gần đây nhất trong bộ nhớ đệm. Ý tưởng chính là: nếu một dữ liệu không được truy cập trong một khoảng thời gian dài, thì khả năng nó sẽ được dùng lại trong tương lai là thấp. Vì thế, khi cache đầy, LRU sẽ loại bỏ phần tử có thời điểm truy cập gần nhất lâu nhất so với các phần tử còn lại.

#### **Ưu điểm:**

- Tận dụng temporal locality tốt: LRU khai thác tốt tính chất temporal locality – dữ liệu được truy cập gần đây có khả năng được dùng lại trong tương lai gần.
- Hiệu suất cao hơn FIFO: Do chỉ loại bỏ các phần tử ít được truy cập gần đây nhất, nên LRU có khả năng giữ lại dữ liệu “nóng” hiệu quả hơn FIFO.
- Được sử dụng phổ biến trong thực tế: Các hệ thống hiện đại như hệ điều hành, CPU cache, và Caching Proxy đều ứng dụng LRU (hoặc biến thể của nó).

#### **Nhược điểm:**

- Chi phí cài đặt cao hơn: LRU yêu cầu theo dõi thứ tự truy cập của các phần tử. Cần cấu trúc dữ liệu như linked list kết hợp với hash map để đảm bảo thao tác thêm/xóa/truy cập đều hiệu quả ( $O(1)$ ), dẫn đến bộ nhớ bổ sung và độ phức tạp cao hơn FIFO.
- Có thể bị “thrashing” trong một số mẫu truy cập đặc biệt: Nếu có một chuỗi truy cập dài mà không lặp lại, LRU sẽ liên tục loại bỏ phần tử cũ và không tận dụng được.

#### **2.6.1.2. Thuật toán ARC (Adaptive Replacement Cache)**

ARC là một thuật toán hiện đại hơn, kết hợp ưu điểm của cả LRU (recency) và LFU (frequency). Nó duy trì hai danh sách chính:

- Một cho các phần tử vừa được truy cập gần đây.
- Một cho các phần tử được truy cập nhiều lần.

ARC sẽ điều chỉnh động độ dài của các danh sách này dựa trên mô hình truy cập hiện tại, từ đó tối ưu hiệu suất cho nhiều loại tải truy cập khác nhau.

#### **Ưu điểm:**

- Thích nghi với các mẫu truy cập khác nhau: ARC tự động điều chỉnh giữa “recency” và “frequency” mà không cần cấu hình trước, giúp hoạt động hiệu quả trên các hệ thống có truy cập không đoán trước được.
- Tận dụng cả temporal và frequency locality: ARC xử lý tốt cả dữ liệu vừa mới được dùng và dữ liệu dùng nhiều lần.
- Tỷ lệ hit cao hơn LRU trong nhiều tình huống thực tế: Đặc biệt trong các hệ thống có mẫu truy cập phức tạp (có cả ngắn hạn lẫn dài hạn), ARC cho kết quả rất tốt.

#### **Nhược điểm:**

- Cài đặt phức tạp hơn: So với FIFO hoặc LRU, ARC cần nhiều cấu trúc dữ liệu hơn và logic điều chỉnh động phức tạp hơn, khó triển khai từ đầu.
- Chi phí xử lý cao hơn một chút: Việc theo dõi và điều chỉnh giữa nhiều danh sách có thể tốn thêm tài nguyên xử lý.
- Ít được tích hợp sẵn: Không có sẵn trong nhiều thư viện chuẩn, và cũng ít được hỗ trợ phần cứng như LRU.

#### **2.6.1.3. Thuật toán thay thế ngẫu nhiên (Random replacement)**

Đúng như tên gọi, khi cache đầy, Random Replacement sẽ chọn ngẫu nhiên một phần tử trong cache để loại bỏ, bất kể tần suất hay thời điểm truy cập.

#### **Ưu điểm:**

- Cực kỳ đơn giản: Việc chọn phần tử ngẫu nhiên không yêu cầu theo dõi bất kỳ thông tin nào về truy cập, frequency, hay recency. Cài đặt rất nhanh, rất gọn.
- Chi phí cực thấp: Không yêu cầu cấu trúc dữ liệu đặc biệt. Trong một số hệ thống tối giản (như thiết bị IoT hoặc nhúng), Random là lựa chọn hợp lý.
- Không bị bias bởi mẫu truy cập: Trong một số tình huống, lựa chọn ngẫu nhiên giúp tránh được các “bẫy” mà các thuật toán như LRU hay FIFO dễ mắc phải (ví dụ như mẫu truy cập dạng tuần hoàn).

#### **Nhược điểm:**

- Hiệu suất không ổn định: Do không có chiến lược nào để giữ lại dữ liệu quan trọng, random replacement có thể loại bỏ ngay cả những phần tử “nóng”.

- Không tận dụng bất kỳ locality nào: Không quan tâm đến temporal hay frequency locality, nên tỉ lệ cache hit thường thấp hơn so với LRU, LFU, ARC.
- Không phù hợp với các hệ thống hiệu năng cao: Trong các hệ thống cần cache tối ưu như cơ sở dữ liệu, CPU cache... Random replacement thường không phải là lựa chọn chính.

## Chương 3. Các loại cache

### 3.1. Hardware Cache:

#### 3.1.1. CPU Cache:

Caching trong bộ nhớ đệm CPU (CPU cache) là một kỹ thuật được sử dụng để tăng tốc độ xử lý của bộ vi xử lý bằng cách lưu trữ tạm thời dữ liệu hoặc lệnh mà CPU thường xuyên sử dụng trong một bộ nhớ nhỏ, nhanh hơn so với bộ nhớ chính (RAM). Bộ nhớ đệm này nằm gần CPU, thường được tích hợp trực tiếp trên cùng con chip, giúp giảm thời gian truy cập dữ liệu và cải thiện hiệu suất tổng thể của hệ thống

##### 3.1.1.1. CPU memory cache là gì?

- Máy tính có nhiều loại bộ nhớ với tốc độ khác nhau:

- Bộ nhớ chính (như ổ cứng hoặc SSD) lưu trữ dữ liệu lớn, chẳng hạn hệ điều hành và chương trình, nhưng truy cập chậm.
- RAM (Random Access Memory) nhanh hơn, lưu trữ tạm thời dữ liệu thường dùng để hỗ trợ hoạt động mượt mà của máy tính.
- CPU cache, nhanh nhất trong số này, nằm gần CPU nhất và lưu dữ liệu mà CPU cần truy cập ngay lập tức.

- CPU memory cache, hay bộ nhớ đệm CPU, là một loại bộ nhớ siêu nhanh nằm trong hoặc gần CPU, giúp tăng tốc độ xử lý dữ liệu. Vào những năm 1980, khi tốc độ CPU tăng nhanh nhưng RAM (bộ nhớ hệ thống) không theo kịp, CPU cache ra đời để giải quyết vấn đề này.

- Bộ nhớ máy tính có hệ thống phân cấp dựa trên tốc độ hoạt động của nó. CPU cache đứng đầu hệ thống phân cấp bộ nhớ nhờ tốc độ vượt trội. Nó sử dụng SRAM (Static RAM), một loại RAM tĩnh có thể chứa dữ liệu mà không cần làm mới liên tục, khác với DRAM (Dynamic RAM) trong RAM hệ thống, điều này khiến SRAM lý tưởng cho cache, giúp CPU xử lý nhanh hơn bằng cách giảm thời gian chờ dữ liệu từ RAM hoặc bộ nhớ chính.

### 3.1.1.2. Cấu trúc của bộ nhớ đệm CPU:

Bộ nhớ cache trong CPU được chia thành nhiều cấp, thường là L1, L2 và L3, mỗi cấp có tốc độ và dung lượng khác nhau. Các cấp này hoạt động theo thứ tự để cung cấp dữ liệu nhanh nhất cho CPU.

- L1 Cache (Level 1): Nhanh nhất và nhỏ nhất, thường có dung lượng từ 8KB đến 64KB. L1 nằm trực tiếp trên chip CPU, được chia thành hai phần: cache lệnh (instructions) và cache dữ liệu (data). Nó phục vụ từng nhân CPU riêng lẻ.
- L2 Cache (Level 2): Lớn hơn và chậm hơn L1, dung lượng thường từ 256KB đến vài MB (1MB, 2MB...). L2 cũng nằm trên chip CPU hoặc rất gần, hỗ trợ một hoặc nhiều nhân CPU.
- L3 Cache (Level 3): Dung lượng lớn nhất (vài MB đến hàng chục MB) nhưng chậm hơn L2. L3 thường được chia sẻ giữa tất cả các nhân trong CPU đa lõi và nằm trên cùng chip.

Khi CPU cần dữ liệu (ví dụ: từ một cú click chuột), nó kiểm tra L1 trước. Nếu không tìm thấy (cache miss), nó chuyển sang L2, rồi L3, và cuối cùng là RAM hoặc ổ cứng nếu cần. Quá trình này diễn ra trong vài nano giây, nhanh đến mức người dùng không nhận ra. Nhờ hệ thống phân cấp này, CPU xử lý hiệu quả hơn, giảm thời gian chờ dữ liệu.

### 3.1.1.3. Cơ chế hoạt động:

Bộ nhớ đệm CPU hoạt động dựa trên nguyên tắc lưu trữ tạm thời các dữ liệu và lệnh mà CPU thường xuyên sử dụng, giúp tăng tốc độ xử lý. Khi CPU cần dữ liệu, nó kiểm tra bộ nhớ đệm trước. Quá trình này có hai trạng thái chính:

- Cache Hit: Nếu dữ liệu đã có trong bộ nhớ đệm (gọi là cache hit), CPU truy cập ngay lập tức mà không cần tìm trong bộ nhớ chính (RAM). Điều này giúp tiết kiệm thời gian vì bộ nhớ đệm nhanh hơn RAM rất nhiều.
- Cache Miss: Nếu dữ liệu không có trong bộ nhớ đệm (cache miss), CPU phải lấy dữ liệu từ cấp bộ nhớ cao hơn (L2, L3) hoặc từ RAM, thậm chí từ ổ cứng nếu cần. Dữ liệu sau đó được sao chép vào bộ nhớ đệm để các lần truy cập sau nhanh hơn. Cache miss làm chậm quá trình xử lý do độ trễ tăng lên khi CPU phải tìm kiếm ở các cấp bộ nhớ xa hơn.

- Cách CPU truy cập bộ nhớ đệm:

CPU kiểm tra dữ liệu theo thứ tự từ gần đến xa:

- Đầu tiên là L1 Cache (nhANH NHẤT, độ trễ thấp nhất, thường vài chục KB), nằm ngay trên chip CPU.
- Nếu không tìm thấy, CPU chuyển sang L2 Cache (lớn hơn, chậm hơn L1), rồi đến L3 Cache (dung lượng lớn nhất, thường vài MB, chia sẻ giữa các nhân CPU).
- Nếu vẫn không có, dữ liệu được lấy từ RAM và cuối cùng là ổ cứng.

Ví dụ, khi bạn mở một chương trình, CPU sẽ cần dữ liệu để xử lý. Nó kiểm tra L1 trước. Nếu không tìm thấy (cache miss), CPU kiểm tra L2, rồi L3. Nếu dữ liệu không có trong các cấp cache, nó được lấy từ RAM (hoặc ổ cứng) và sao chép vào các cấp cache (thường bắt đầu từ L3, sau đó có thể vào L2 và L1) để các lần truy cập sau nhanh hơn.”

#### **3.1.1.4. Tầm quan trọng của bộ nhớ đệm CPU:**

Bộ nhớ cache đóng vai trò quan trọng trong việc nâng cao hiệu suất hệ thống máy tính bằng cách giảm độ trễ truy cập dữ liệu và tăng tốc độ xử lý của CPU. Dưới đây là các lợi ích chính của bộ nhớ cache:

- Tăng tốc độ truy cập dữ liệu: Bộ nhớ cache lưu trữ các dữ liệu và lệnh mà CPU thường xuyên sử dụng, giúp giảm thời gian truy cập so với việc lấy từ RAM. Điều này đặc biệt quan trọng với các ứng dụng đòi hỏi hiệu năng cao như xử lý video, đồ họa 3D, và game, nơi tốc độ xử lý nhanh là yếu tố then chốt.
- Giảm tải cho bộ nhớ chính: Bằng cách lưu trữ dữ liệu thường dùng, bộ nhớ cache giảm áp lực lên RAM, cho phép RAM xử lý các tác vụ khác hiệu quả hơn. Điều này cải thiện độ trễ và tăng hiệu suất tổng thể của hệ thống.
- Tối ưu hóa tài nguyên: Các công nghệ như Intel Smart Cache cho phép các lõi CPU trong hệ thống đa lõi chia sẻ bộ nhớ cache một cách thông minh, đảm bảo sử dụng tài nguyên hiệu quả và tăng hiệu suất xử lý.
- Tiết kiệm điện năng: Bộ nhớ cache giảm số lần CPU phải truy cập RAM (một quá trình tiêu tốn nhiều năng lượng hơn), từ đó giúp tiết kiệm điện, đặc biệt hữu ích trong các thiết bị di động hoặc máy tính xách tay.
- Hỗ trợ đa nhiệm và cải thiện trải nghiệm người dùng: Trong môi trường đa nhiệm, bộ nhớ cache giúp CPU xử lý nhiều tác vụ đồng thời mà không bị chậm lại. Điều này rất quan trọng với các máy chủ, hệ thống xử lý dữ liệu lớn, hoặc các tác vụ hàng ngày như mở ứng dụng, duyệt web, và chơi game, mang lại trải nghiệm mượt mà hơn.

Mặc dù có nhiều lợi ích, bộ nhớ cache cũng tồn tại một số hạn chế:

- Dung lượng hạn chế: Bộ nhớ cache (thường chỉ vài KB đến vài MB) nhỏ hơn nhiều so với RAM, dẫn đến hiện tượng cache miss khi dữ liệu không có sẵn. Điều này gây độ trễ vì CPU phải truy cập RAM hoặc ổ cứng. Để khắc phục, các nhà sản xuất có thể tăng dung lượng cache (như L3 cache lên đến 128MB trong một số CPU hiện đại) hoặc sử dụng thuật toán dự đoán dữ liệu thông minh hơn (prefetching) để giảm cache miss.
- Chi phí và diện tích vật lý: Bộ nhớ cache sử dụng SRAM (Static RAM), nhanh nhưng đắt hơn DRAM (Dynamic RAM) trong RAM. Tăng dung lượng hoặc cải thiện cấu trúc cache sẽ làm tăng chi phí sản xuất và diện tích trên chip, đặc biệt trong các thiết bị nhúng hoặc di động. Để giải quyết, các nhà thiết kế thường cân bằng giữa hiệu suất và chi phí, ưu tiên tối ưu hóa thuật toán cache thay vì tăng kích thước vật lý.
- Tính không nhất quán dữ liệu (cache coherence): Vì dữ liệu trong cache là bản sao tạm thời từ RAM, nếu không được cập nhật đúng cách, dữ liệu có thể trở nên lỗi thời, đặc biệt trong hệ thống đa lõi nơi nhiều lõi CPU chia sẻ dữ liệu. Ví dụ, nếu một lõi cập nhật dữ liệu trong cache của nó mà không đồng bộ với các lõi khác, sẽ dẫn đến lỗi. Để khắc phục, các giao thức như MESI (Modified, Exclusive, Shared, Invalid) được sử dụng để đảm bảo tính nhất quán dữ liệu giữa các cache và RAM.

#### **3.1.1.5. Các công nghệ và xu hướng hiện đại:**

Bộ nhớ cache CPU ngày càng được cải tiến để đáp ứng nhu cầu hiệu năng cao của các hệ thống máy tính hiện đại. Dưới đây là các công nghệ và xu hướng nổi bật:

##### **3.1.1.5.1. Bộ nhớ cache đa tầng (Multi-Level Cache) và L4 Cache:**

Các CPU cao cấp hiện nay đã tích hợp thêm L4 cache, bổ sung một tầng bộ nhớ đệm mới ngoài L1, L2, L3. L4 cache thường có dung lượng lớn hơn L3 (lên đến hàng chục MB) và được sử dụng để lưu trữ dữ liệu cho các ứng dụng đòi hỏi tài nguyên cao, như xử lý đồ họa 3D hoặc máy học. Ví dụ, một số CPU của Intel (như dòng Xeon) sử dụng L4 cache làm bộ nhớ đệm eDRAM, giúp giảm độ trễ khi truy cập RAM và tăng hiệu suất tổng thể.

##### **3.1.1.5.2. Bộ nhớ Non-Volatile (NVM) cho cache:**

Bộ nhớ không khả biến (Non-Volatile Memory - NVM), như 3D XPoint (Intel Optane), đang được nghiên cứu để sử dụng trong L4 cache. NVM có ưu điểm là dung lượng lớn, độ bền cao, và giá thành rẻ hơn DRAM truyền thống. Tuy nhiên, hạn chế của NVM là tốc độ truy cập chậm hơn và độ trễ cao hơn so với SRAM (dùng trong L1-L3). Để khắc phục, các nhà thiết kế thường kết hợp NVM với SRAM trong cấu trúc lai (hybrid cache), tận dụng ưu điểm của cả hai.



#### **3.1.1.5.3. Công nghệ Smart Cache:**

Được Intel tiên phong, Smart Cache cho phép các lõi CPU trong hệ thống đa lõi chia sẻ bộ nhớ cache (thường là L3) một cách linh hoạt. Thay vì mỗi lõi có bộ nhớ đệm riêng, Smart Cache phân bổ tài nguyên động dựa trên nhu cầu, giảm lãng phí và cải thiện hiệu suất đa nhiệm. Công nghệ này đặc biệt hữu ích trong các tác vụ như xử lý dữ liệu lớn hoặc chơi game, nơi nhiều lõi cần truy cập dữ liệu chung.

#### **3.1.1.5.4. Non-Uniform Cache Access (NUCA):**

NUCA là kỹ thuật tối ưu hóa cho các bộ nhớ cache lớn trong CPU đa lõi. NUCA chia bộ nhớ cache thành các vùng (banks) với thời gian truy cập khác nhau, tùy thuộc vào khoảng cách vật lý đến lõi CPU. Dữ liệu thường xuyên sử dụng được lưu ở vùng gần lõi (truy cập nhanh), trong khi dữ liệu ít dùng hơn được lưu ở vùng xa hơn (truy cập chậm hơn). NUCA giúp giảm độ trễ trung bình và tăng hiệu suất, nhưng nhược điểm là tăng độ phức tạp trong thiết kế và quản lý cache.

#### **3.1.1.5.5. Giao thức Cache Coherence (MESI và MOESI):**

Trong hệ thống đa lõi, **Cache Coherence Protocols** đảm bảo tính nhất quán dữ liệu giữa các lõi CPU, tránh tình trạng một lõi đọc dữ liệu lỗi thời từ cache của nó. Hai giao thức phổ biến là:

- **MESI (Modified, Exclusive, Shared, Invalid):**
  - **Modified:** Dữ liệu trong cache đã được sửa đổi và khác với bản sao trong RAM.
  - **Exclusive:** Dữ liệu chỉ có trong cache của lõi này, chưa được chia sẻ.
  - **Shared:** Dữ liệu được chia sẻ giữa nhiều lõi, các bản sao giống nhau.
  - **Invalid:** Dữ liệu không hợp lệ, cần cập nhật từ RAM.
- **MOESI (Modified, Owned, Exclusive, Shared, Invalid):**

Tương tự MESI, nhưng bổ sung trạng thái Owned: Lõi này sở hữu dữ liệu và có trách nhiệm cập nhật RAM khi cần, cho phép chia sẻ dữ liệu đã sửa đổi mà không cần ghi ngay vào RAM, giảm tải cho bộ nhớ chính.

### **3.1.2. GPU cache:**

#### **3.1.2.1. Bộ nhớ đệm GPU:**

- Bên cạnh bộ nhớ cache CPU, bộ nhớ cache GPU cũng đóng vai trò quan trọng trong việc tối ưu hóa hiệu năng của các hệ thống xử lý đồ họa và tính toán song song. GPU (Graphics Processing Unit), hay còn gọi là bộ xử lý đồ họa, là thành phần chuyên dụng trong máy tính, chịu trách nhiệm xử lý các tác vụ đồ họa phức tạp và các phép tính song song.

- GPU cache là bộ nhớ đệm tốc độ cao được tích hợp bên trong GPU, lưu trữ dữ liệu và lệnh mà GPU thường xuyên truy cập. Bằng cách giảm thời gian truy xuất dữ liệu từ bộ nhớ chính (thường là VRAM hoặc RAM hệ thống, vốn chậm hơn nhiều), GPU cache giúp tăng tốc độ xử lý, cải thiện đáng kể hiệu suất đồ họa và tính toán.

#### **3.1.2.2. Cấu trúc của bộ nhớ đệm CPU:**

Bộ nhớ cache trong GPU cũng được thiết kế theo kiến trúc đa tầng, tương tự CPU, nhưng được tối ưu hóa cho xử lý đồ họa và tính toán song song. GPU cache thường bao gồm các cấp L1 và L2, cùng với các loại cache chuyên biệt như Texture Cache và Constant Cache, nhằm đáp ứng nhu cầu truy cập dữ liệu nhanh chóng của hàng nghìn luồng xử lý đồng thời.

- L1 Cache: Nằm gần các lõi xử lý của GPU, được gọi là Streaming Multiprocessors (SMs) trong GPU NVIDIA hoặc Compute Units trong GPU AMD. Mỗi SM thường có L1 cache riêng (dung lượng khoảng 128KB trong GPU NVIDIA RTX 4090), lưu trữ dữ liệu và lệnh thường xuyên sử dụng. L1 cache giúp giảm độ trễ khi truy cập dữ liệu cục bộ, đặc biệt trong các tác vụ tính toán song song như render khung hình hoặc huấn luyện mô hình AI.
- L2 Cache: Lớn hơn L1 (thường vài MB, ví dụ: 72MB trong RTX 4090), L2 cache được chia sẻ giữa tất cả các SMs trong GPU. L2 cache đóng vai trò trung gian, lưu trữ dữ liệu không có trong L1, giúp giảm độ trễ khi truy cập bộ nhớ đồ họa (VRAM) hoặc RAM hệ thống. Với dung lượng lớn, L2 cache đặc biệt hữu ích trong các tác vụ như ray tracing, nơi cần truy cập lượng dữ liệu lớn từ VRAM.
- Texture Cache: Là một phần chuyên biệt (thường tích hợp trong L1 hoặc L2), được tối ưu hóa để xử lý dữ liệu đồ họa như texture và hình ảnh. Texture Cache hỗ trợ truy cập ngẫu nhiên nhanh chóng, đặc biệt với các truy vấn không gian hai chiều (2D spatial queries), giúp tăng hiệu suất render trong game hoặc ứng dụng 3D. Nhiều GPU hiện đại (như dòng NVIDIA GeForce) hỗ trợ nén texture, giảm lượng dữ liệu truyền tải, tiết kiệm băng thông VRAM và cải thiện hiệu suất tổng thể.
- Constant Cache: Cũng là một phần chuyên biệt của L1 cache, dùng để lưu trữ các hằng số không thay đổi trong quá trình xử lý (ví dụ: thông số ánh sáng trong render đồ họa). Vì dữ liệu trong Constant Cache không bị ghi đè, việc truy cập rất nhanh và không gây xung đột, giúp GPU quản lý các hằng số hiệu quả trong các tác vụ như tính toán shader.

#### **So sánh với CPU cache:**

Khác với CPU cache (tập trung vào giảm độ trễ cho các tác vụ tuần tự), GPU cache được thiết kế để tối ưu hóa băng thông và hỗ trợ tính toán song song. GPU cache thường có dung lượng lớn hơn (đặc biệt là L2) để xử lý hàng nghìn luồng dữ liệu cùng lúc, nhưng độ trễ trung bình cao hơn do số lượng truy cập đồng thời lớn.

Ví dụ, trong khi L1 cache của CPU có độ trễ khoảng 1-2 chu kỳ, L1 cache của GPU có thể lên đến 20-30 chu kỳ, nhưng bù lại bằng thông cao hơn nhiều để đáp ứng nhu cầu đồ họa.

### **Xu hướng hiện đại:**

Các GPU mới như NVIDIA Ada Lovelace (RTX 40-series) tăng dung lượng L2 cache và tích hợp các thuật toán dự đoán dữ liệu (prefetching) để giảm cache miss. Ngoài ra, một số GPU thử nghiệm L0 cache (dành riêng cho texture hoặc instruction), giúp tăng hiệu suất trong các tác vụ đồ họa phức tạp như ray tracing hoặc xử lý video 8K.

### **3.1.2.3. Cơ chế hoạt động:**

GPU cache hoạt động dựa trên **nguyên lý địa phương (locality principle)**, một khái niệm quan trọng trong kiến trúc máy tính giúp tối ưu hóa hiệu suất bộ nhớ đệm.

Nguyên lý này bao gồm hai khía cạnh chính: **địa phương không gian (spatial locality)** và **địa phương thời gian (temporal locality)**, được GPU tận dụng để tăng tốc xử lý đồ họa và tính toán song song.

- **Spatial Locality (Địa phương không gian):** Nếu một địa chỉ bộ nhớ được truy cập, các địa chỉ lân cận có khả năng được truy cập tiếp theo. Trong xử lý đồ họa, GPU thường truy cập các pixel hoặc texel gần nhau, ví dụ khi render hình ảnh và áp dụng bộ lọc màu sắc. Để tối ưu hóa việc truy cập, Texture Cache được thiết kế để lưu trữ các khối texture lớn, giảm số lần truy cập VRAM (bộ nhớ đồ họa chính của GPU), từ đó tăng hiệu suất render.
- **Temporal Locality (Địa phương thời gian):** Nếu một địa chỉ bộ nhớ được truy cập, nó có khả năng được truy cập lại trong tương lai gần. GPU tận dụng điều này bằng cách lưu trữ các hằng số (constant data) trong Constant Cache, ví dụ như thông số ánh sáng trong shader. Vì các hằng số không thay đổi, Constant Cache cho phép truy cập nhanh và hiệu quả, giảm độ trễ.

### **Luồng truy cập dữ liệu trong GPU cache:**

Khi một chương trình đồ họa yêu cầu dữ liệu, GPU kiểm tra L1 cache trước (nằm gần các lõi xử lý - Streaming Multiprocessors). Nếu dữ liệu có sẵn (cache hit), GPU truy cập ngay lập tức. Nếu không (cache miss), GPU kiểm tra L2 cache, vốn lớn hơn và được chia sẻ giữa các SMs. Nếu L2 cũng không có, dữ liệu được lấy từ VRAM (bộ nhớ đồ họa chính, chậm hơn nhiều). Sau đó, dữ liệu này được sao chép vào L2 và L1 để các lần truy cập sau nhanh hơn. Quá trình này diễn ra trong vài nano giây, đảm bảo hiệu suất cao cho các tác vụ đồ họa.

### **Quản lý GPU cache:**

Để tối ưu hóa việc sử dụng cache, GPU áp dụng các thuật toán quản lý dữ liệu:

- LRU (Least Recently Used): Loại bỏ dữ liệu ít được truy cập nhất để nhường chỗ cho dữ liệu mới.
- FIFO (First In, First Out): Loại bỏ dữ liệu cũ nhất trong cache.
- PLRU (Pseudo-Least Recently Used): Kết hợp ưu điểm của LRU và FIFO, sử dụng các bit trạng thái để ước lượng dữ liệu ít dùng, giảm chi phí tính toán so với LRU thuần túy.
- Adaptive Cache Replacement: Điều chỉnh thuật toán dựa trên mẫu truy cập dữ liệu, ví dụ ưu tiên giữ texture trong Texture Cache nếu GPU đang render nhiều khung hình.

#### **3.1.2.4. Tầm quan trọng của bộ nhớ đệm GPU:**

Bộ nhớ cache GPU đóng vai trò quan trọng trong việc nâng cao hiệu suất xử lý đồ họa và tính toán song song, giúp giảm độ trễ truy cập dữ liệu và tăng tốc độ hoạt động của GPU. Nhờ cache, các ứng dụng yêu cầu hiệu năng cao như đồ họa, xử lý video, và game chạy mượt mà và nhanh hơn. Dưới đây là các lợi ích và hạn chế của GPU, với vai trò của bộ nhớ cache được làm rõ:

#### **Lợi ích của GPU và vai trò của bộ nhớ cache:**

- Tăng tốc xử lý song song: GPU có hàng nghìn lõi xử lý (ví dụ: NVIDIA RTX 4090 có hơn 16.000 lõi CUDA), cho phép thực hiện nhiều tác vụ đồng thời. Bộ nhớ cache GPU (L1, L2, Texture Cache) lưu trữ dữ liệu thường xuyên truy cập, như ma trận trong học máy hoặc texture trong render, giúp giảm độ trễ khi truy cập VRAM. Điều này rất quan trọng trong các ứng dụng như phân tích dữ liệu, tính toán khoa học, và huấn luyện mô hình deep learning (ví dụ: nhận dạng khuôn mặt, thị giác máy tính).
- Render đồ họa nhanh chóng: GPU chuyên xử lý đồ họa, render hình ảnh 3D và hiệu ứng chân thực (như bóng động, phản xạ ánh sáng, giảm nhiễu) một cách mượt mà. Texture Cache lưu trữ các khối texture lớn, giảm số lần truy cập VRAM, từ đó tăng tốc render trong game (như Cyberpunk 2077 ở 4K) và ứng dụng thiết kế 3D. Các công nghệ như NVIDIA CUDA, AMD Stream, và OpenCL tận dụng khả năng tính toán song song của GPU, kết hợp với cache để tối ưu hiệu suất.
- Mô phỏng phức tạp: GPU hỗ trợ mô phỏng vật lý (như động lực học chất lỏng, mô phỏng phân tử) nhờ khả năng tính toán ma trận nhanh. L1 và L2 cache lưu trữ dữ liệu tạm thời, giảm độ trễ khi xử lý các phép tính số học lớn, giúp các nhà khoa học thực hiện nghiên cứu tiên tiến.
- Trải nghiệm VR/AR và chơi game: GPU mang lại hình ảnh sống động trong game và trải nghiệm VR/AR chân thực, hỗ trợ công nghệ như 8K và HDR. Texture Cache và Constant Cache đảm bảo truy cập nhanh texture và hằng số, giảm độ trễ khung hình, đặc biệt trong các game AAA hoặc ứng dụng VR.

- Chi phí hiệu quả và khả năng mở rộng: GPU thực hiện các tác vụ tính toán nặng với chi phí phần cứng thấp hơn so với việc dùng nhiều CPU. Bộ nhớ cache giảm tải cho VRAM, giúp GPU xử lý hiệu quả hơn. Hệ thống có thể mở rộng bằng cách thêm GPU mà không cần thay đổi kiến trúc.

*Render các phần mềm 3d*

*Xử lý tác vụ trong các tựa game AAA*

### **Hạn chế của GPU và cách khắc phục:**

- Tiêu thụ năng lượng cao: GPU hiệu suất cao (như NVIDIA A100) tiêu thụ nhiều điện (lên đến 400W), tăng chi phí vận hành và ảnh hưởng môi trường. Các nhà sản xuất đang phát triển công nghệ tiết kiệm năng lượng, như NVIDIA Max-Q, để giảm tiêu thụ điện trong GPU di động.
- Khó lập trình: Lập trình GPU đòi hỏi kiến thức về CUDA, OpenCL, hoặc Vulkan, và hiểu biết về kiến trúc song song, khiến việc phát triển và debug phức tạp. Các công cụ như NVIDIA Nsight và AMD ROCm đang được cải thiện để hỗ trợ lập trình viên, cùng với sự phát triển của các framework như TensorFlow và PyTorch, giúp đơn giản hóa việc lập trình GPU.
- Kích thước và khả năng tương thích: GPU thường lớn hơn CPU, làm tăng kích thước hệ thống và giảm tính di động. Một số phần mềm không tận dụng được GPU, dẫn đến hiệu suất không tối ưu. Các nhà sản xuất đang phát triển GPU tích hợp (như AMD APUs) để giảm kích thước, và cải thiện khả năng tương thích qua các API chuẩn như DirectX và Vulkan.
- Hạn chế trong tác vụ tuần tự: GPU không hiệu quả trong các tác vụ tuần tự hoặc phụ thuộc dữ liệu (data-dependent tasks), vì được thiết kế cho tính toán song song. Để khắc phục, các hệ thống thường kết hợp CPU và GPU, với CPU xử lý tác vụ tuần tự và GPU tập trung vào tác vụ song song.

### **3.1.2.5. Các công nghệ và xu hướng hiện đại:**

Công nghệ và xu hướng hiện đại áp dụng cho GPU đang phát triển nhanh chóng, mở ra nhiều cơ hội mới trong tính toán và đồ họa. Bộ nhớ cache GPU đóng vai trò quan trọng trong việc hỗ trợ các công nghệ này, giúp giảm độ trễ truy cập dữ liệu và tăng hiệu suất xử lý. Dưới đây là các xu hướng nổi bật:

#### **3.1.2.5.1. Tăng tốc Machine Learning và Deep Learning:**

GPU là lựa chọn hàng đầu cho huấn luyện mô hình học máy (ML) và học sâu (DL) nhờ khả năng tính toán song song. Trong ML/DL, các phép tính ma trận lớn (như backpropagation, stochastic gradient descent) được thực hiện nhanh hơn hàng chục lần so với CPU. Bộ nhớ cache GPU (L1, L2) lưu trữ trọng số mô hình và dữ liệu huấn luyện, giảm độ trễ khi truy cập VRAM. Ví dụ, trong NVIDIA A100, Tensor Cores

(đơn vị chuyên biệt cho tính toán ma trận) kết hợp với L2 cache 141MB giúp tăng tốc huấn luyện mô hình AI gấp 20 lần so với thế hệ trước. Cache cũng lưu trữ kết quả dự đoán để tái sử dụng, hữu ích trong các ứng dụng như nhận dạng khuôn mặt hoặc thị giác máy tính.

#### **3.1.2.5.2. Mixed-Precision Arithmetic (MPA):**

MPA sử dụng các định dạng số có độ chính xác khác nhau (như FP16, FP32) để tăng tốc tính toán trong deep learning mà không làm giảm độ chính xác mô hình.

Cache GPU hỗ trợ MPA bằng cách lưu trữ dữ liệu ở định dạng nén (như FP16), giảm băng thông VRAM và tăng hiệu suất. Ví dụ, khi huấn luyện mô hình trên GPU NVIDIA H100, MPA kết hợp với cache giúp giảm thời gian huấn luyện đến 50% so với tính toán toàn chính xác (FP32).

#### **3.1.2.5.3. Real-Time Ray Tracing và Rendering:**

Ray tracing mô phỏng cách ánh sáng tương tác trong môi trường 3D, tạo ra hình ảnh chân thực với bóng, phản xạ, và khúc xạ chính xác. Real-time ray tracing, được hỗ trợ bởi NVIDIA RTX (dòng RTX 40-series) và AMD Radeon RX 6000 series, mang lại trải nghiệm đồ họa sống động trong game và ứng dụng 3D.

Bộ nhớ cache GPU (đặc biệt Texture Cache) lưu trữ dữ liệu mô hình 3D và texture, giảm độ trễ khi truy cập VRAM. Cache cũng lưu trữ kết quả render của các khung hình trước để tái sử dụng trong các khung hình tiếp theo, đặc biệt hiệu quả khi camera di chuyển chậm (như trong game Control với ray tracing bật).

#### **3.1.2.5.4. AI-Powered Rendering:**

AI cải thiện quá trình rendering bằng cách tự động hóa các tác vụ như thiết lập ánh sáng, áp dụng vật liệu, hoặc tối ưu hóa hiệu suất. Ví dụ, NVIDIA DLSS (Deep Learning Super Sampling) sử dụng AI để upscale hình ảnh từ độ phân giải thấp lên cao (như 1080p lên 4K), vừa tăng chất lượng hình ảnh vừa giảm tải cho GPU. Cache GPU lưu trữ các trọng số mô hình AI và dữ liệu khung hình, giúp tăng tốc quá trình upscale và render, mang lại trải nghiệm mượt mà trong game.

#### **3.1.2.5.5. Hybrid Rendering và Mô phỏng Vật lý:**

- Hybrid Rendering: Kết hợp ray tracing với các kỹ thuật truyền thống như rasterization để cân bằng giữa chất lượng hình ảnh và hiệu suất. Cache GPU lưu trữ dữ liệu ánh sáng và texture, giảm độ trễ trong các cảnh phức tạp.
- GPU-Accelerated Physics Simulation: GPU tăng tốc mô phỏng vật lý (như động lực học chất lỏng, mô phỏng hạt nhân) nhờ tính toán song song. L1 cache lưu trữ dữ liệu tạm thời (như vị trí hạt), giúp GPU xử lý hàng triệu phép tính mỗi giây, hữu ích trong nghiên cứu khoa học và thiết kế kỹ thuật.

## 3.2. Disk Cache:

Bộ nhớ đệm đĩa (Disk Cache) là một thành phần quan trọng trong hệ thống lưu trữ, giúp tăng tốc độ đọc và ghi dữ liệu từ ổ đĩa cứng (HDD) hoặc ổ SSD.

### 3.2.1. Khái niệm:

Disk Cache có thể là bộ nhớ tích hợp trên ổ đĩa (onboard cache, thường 8MB-256MB trong HDD hiện đại) hoặc một phần RAM được hệ điều hành sử dụng để làm bộ nhớ đệm. Bằng cách lưu trữ dữ liệu đã truy cập gần đây và dữ liệu liên kế có khả năng được truy cập tiếp theo, Disk Cache giảm độ trễ truy xuất từ ổ đĩa (chậm hơn nhiều so với RAM), từ đó cải thiện hiệu suất hệ thống.

### 3.2.2. Cách disk cache hoạt động:

Disk Cache hoạt động dựa trên hai cơ chế chính:

- Đọc trước (Read-Ahead): Khi hệ thống đọc một khối dữ liệu từ ổ đĩa, các khối liên kế cũng được tải vào cache, dự đoán rằng chúng sẽ được truy cập tiếp theo (dựa trên nguyên lý địa phương không gian - spatial locality).
- Ghi sau (Write-Behind): Dữ liệu ghi được lưu vào cache trước, sau đó được ghi xuống ổ đĩa theo cách tối ưu (ví dụ: gom nhóm các thao tác ghi để giảm số lần truy cập ổ đĩa). Điều này tận dụng tốc độ nhanh của RAM, giảm thời gian chờ.

#### *Disk cache*

Khi dữ liệu trong cache bị thay đổi, hệ thống sử dụng hai chính sách để đồng bộ với ổ đĩa:

- Write-Through: Dữ liệu được ghi đồng thời vào cache và ổ đĩa, đảm bảo tính toàn vẹn nhưng chậm hơn.

#### *Write-through*

- Write-Back: Dữ liệu chỉ ghi vào cache trước, sau đó được đồng bộ với ổ đĩa sau, nhanh hơn nhưng có nguy cơ mất dữ liệu nếu mất điện

### 3.2.3. Kiến trúc của disk cache:

Disk Cache bao gồm các thành phần chính:

- Read Cache: Lưu trữ dữ liệu đã đọc gần đây từ ổ đĩa, giúp tăng tốc các yêu cầu đọc tiếp theo.
- Write Cache: Đệm các thao tác ghi, gom nhóm và tối ưu hóa trước khi ghi xuống ổ đĩa, cải thiện hiệu suất ghi.
- Memory Buffer:

- ▶ **RAM Cache:** Sử dụng RAM để lưu trữ dữ liệu tạm thời, nhanh nhưng mất dữ liệu nếu mất điện.
- ▶ **Non-Volatile Memory (NVM):** Một số ổ đĩa cao cấp (như SSD NVMe) dùng NVM để bảo vệ dữ liệu trong trường hợp mất điện.
- **Cache Controller:** Quản lý việc lưu trữ và truy xuất dữ liệu, sử dụng các thuật toán như LRU (Least Recently Used) (loại bỏ dữ liệu ít dùng nhất) và FIFO (First In, First Out) (loại bỏ dữ liệu cũ nhất) để tối ưu không gian cache.

#### 3.2.4. Tác dụng của disk cache:

Bộ nhớ đệm đĩa (Disk Cache) mang lại nhiều lợi ích quan trọng, giúp tăng hiệu suất hệ thống và cải thiện trải nghiệm người dùng. Dưới đây là các lợi ích chính, cùng với vai trò cụ thể của Disk Cache:

##### 3.2.4.1. Tăng tốc độ truy cập dữ liệu:

- **Giảm độ trễ đọc:** Disk Cache lưu trữ dữ liệu đã truy cập gần đây trong RAM hoặc bộ nhớ tích hợp trên ổ đĩa (onboard cache), nhanh hơn nhiều so với truy xuất trực tiếp từ HDD/SSD. Ví dụ, với một HDD có tốc độ đọc 200MB/s, Disk Cache (thường 64MB-256MB) có thể giảm thời gian truy cập từ 10ms xuống dưới 1ms nếu dữ liệu đã có trong cache (cache hit).
- **Cơ chế đọc trước (Read-Ahead):** Disk Cache dự đoán và tải trước các khối dữ liệu liên kề, dựa trên nguyên lý địa phương không gian (spatial locality). Điều này đặc biệt hữu ích trong HDD khi đọc các tệp lớn (như video), giảm thời gian chờ.

##### 3.2.4.2. Cải thiện hiệu suất ghi:

- **Ghi nhanh hơn:** Write Cache lưu dữ liệu ghi vào bộ nhớ đệm trước, cho phép hệ thống xác nhận thao tác ghi hoàn tất ngay lập tức từ góc độ ứng dụng, trong khi dữ liệu được ghi xuống ổ đĩa sau (write-behind). Điều này tăng tốc độ ghi, đặc biệt trong các tác vụ ghi liên tục như sao chép tệp lớn.
- **Quản lý ghi hiệu quả:** Disk Cache gom nhóm các thao tác ghi nhỏ lẻ, tối ưu hóa việc ghi xuống ổ đĩa, giảm số lần truy cập vật lý. Với SSD, điều này giúp giảm chu kỳ ghi/xóa NAND, kéo dài tuổi thọ ổ.

##### 3.2.4.3. Tăng hiệu suất hệ thống tổng thể:

- **Giảm tải cho ổ đĩa:** Bằng cách giảm số lần truy cập trực tiếp đến HDD/SSD, Disk Cache giảm tải cho ổ đĩa, cải thiện hiệu suất tổng thể và tối ưu hóa sử dụng tài nguyên hệ thống (như RAM).
- **Hiệu quả với SSD:** Trong SSD NVMe (như Samsung 990 Pro), Disk Cache (thường là DRAM 1-2GB) giúp tăng tốc độ đọc/ghi lên đến 7.450MB/s, đặc biệt khi xử lý dữ liệu lớn như chỉnh sửa video 4K.



#### 3.2.4.4. Cải thiện trải nghiệm người dùng:

- **Phản hồi nhanh hơn:** Thời gian truy cập dữ liệu nhanh hơn giúp hệ thống phản hồi mượt mà, đặc biệt trong các ứng dụng yêu cầu độ trễ thấp như cơ sở dữ liệu trực tuyến (OLTP) hoặc khởi động hệ điều hành (giảm thời gian khởi động Windows từ 30 giây xuống 15 giây).
- **Ứng dụng thực tế:** Trong chỉnh sửa video, Disk Cache lưu trữ các khung hình đã đọc, tăng tốc độ phát lại và render, mang lại trải nghiệm liền mạch.

#### 3.2.4.5. Tăng độ an toàn dữ liệu (trong một số trường hợp):

- **Bảo vệ dữ liệu tạm thời:** Một số ổ đĩa cao cấp (như SSD doanh nghiệp) dùng non-volatile memory (NVM) trong Write Cache để bảo vệ dữ liệu trong trường hợp mất điện, giảm nguy cơ mất dữ liệu trong write-back cache. Ví dụ, SSD Intel Optane dùng 3D XPoint làm NVM, đảm bảo tính toàn vẹn dữ liệu

#### Hạn chế của Disk Cache:

- **Nguy cơ mất dữ liệu:** Trong write-back cache, dữ liệu trong RAM có thể bị mất nếu mất điện. Giải pháp là dùng NVM hoặc pin dự phòng (như trong SSD doanh nghiệp).
- **Dung lượng hạn chế:** Cache nhỏ (vài MB đến vài GB) có thể bị đầy, dẫn đến cache miss. Các thuật toán như LRU và Adaptive Cache Replacement giúp ưu tiên dữ liệu quan trọng.

#### 3.2.5. Tương lai của Disk Cache:

Sự phát triển của công nghệ lưu trữ và nhu cầu ngày càng cao từ các ứng dụng hiện đại đang định hình tương lai của bộ nhớ đệm đĩa (Disk Cache). Dưới đây là các xu hướng nổi bật và vai trò của Disk Cache trong bối cảnh mới:

##### 3.2.5.1. Tăng hiệu suất với SSD và NVMe:

Sự phổ biến của ổ SSD, đặc biệt là SSD NVMe (Non-Volatile Memory Express), đang nâng cao hiệu suất của Disk Cache. NVMe cung cấp tốc độ đọc/ghi vượt trội (lên đến 7.450MB/s trong Samsung 990 Pro, so với 200MB/s của HDD) và độ trễ thấp hơn nhiều (dưới 20μs so với 4ms của HDD). Điều này làm cho Disk Cache hiệu quả hơn trong việc giảm độ trễ, đặc biệt khi xử lý dữ liệu lớn. Tuy nhiên, vì SSD đã rất nhanh, vai trò của cơ chế đọc trước (read-ahead) giảm đi, trong khi Write Cache vẫn quan trọng để tối ưu hóa ghi và kéo dài tuổi thọ NAND.

##### 3.2.5.2. Tích hợp bộ nhớ không bay hơi (NVM):

Công nghệ bộ nhớ không bay hơi (NVM) như 3D XPoint (phát triển bởi Intel và Micron) và Intel Optane đang mở ra khả năng mới cho Disk Cache. 3D XPoint có tốc độ gần bằng DRAM (độ trễ ~100ns so với 15ns của DRAM) nhưng không mất dữ liệu khi mất điện, phù hợp để làm bộ nhớ đệm trung gian giữa RAM và SSD.

Ví dụ, Intel Optane được dùng trong SSD doanh nghiệp làm Write Cache, đảm bảo tính toàn vẹn dữ liệu trong trường hợp mất điện, đồng thời tăng tốc độ truy cập lên đến 2.000MB/s. Trong tương lai, NVM có thể thay thế DRAM trong Disk Cache, giảm chi phí và tăng độ bền.

### 3.2.5.3. Quản lý thông minh với AI và máy học:

Trí tuệ nhân tạo (AI) và máy học (ML) đang được tích hợp để tối ưu hóa quản lý Disk Cache. Các thuật toán AI có thể dự đoán dữ liệu nào sẽ được truy cập (dựa trên mẫu truy cập), ưu tiên lưu trữ chúng trong cache, giảm tỷ lệ cache miss.

Ví dụ, trong các hệ thống big data, AI có thể dự đoán các khối dữ liệu thường xuyên truy cập (như kết quả truy vấn cơ sở dữ liệu), tăng tốc độ xử lý lên đến 30%. Ngoài ra, ML có thể điều chỉnh thuật toán thay thế cache (như LRU, FIFO) theo thời gian thực, tối ưu hóa hiệu suất trong các ứng dụng như phân tích dữ liệu lớn hoặc máy chủ web.

### 3.2.5.4. Hỗ trợ xử lý dữ liệu lớn (Big Data):

Các ứng dụng big data yêu cầu truy xuất và lưu trữ dữ liệu lớn một cách nhanh chóng. Disk Cache sẽ đóng vai trò quan trọng trong việc giảm tải cho hệ thống lưu trữ chính (HDD/SSD), lưu trữ tạm thời các khối dữ liệu thường xuyên truy cập (như tệp log, kết quả truy vấn).

Ví dụ, trong một cụm Hadoop, Disk Cache có thể giảm thời gian truy vấn từ 10 giây xuống 2 giây bằng cách lưu trữ dữ liệu trung gian trong RAM.

### 3.2.5.5. Xu hướng mới:

- **Host Memory Buffer (HMB):** SSD NVMe không có DRAM (DRAM-less) ngày càng phổ biến trong thiết bị giá rẻ. HMB cho phép SSD dùng một phần RAM hệ thống (thường 64MB-256MB) làm Disk Cache, tăng hiệu suất đọc/ghi mà không cần DRAM tích hợp. Ví dụ, SSD Kingston NV2 dùng HMB đạt tốc độ đọc 3.500MB/s, gần bằng SSD có DRAM.
- **Tích hợp với lưu trữ đám mây:** Disk Cache có thể được mở rộng để lưu trữ dữ liệu từ đám mây, giảm độ trễ khi truy cập dữ liệu từ xa. Ví dụ, các hệ thống hybrid cloud dùng Disk Cache để lưu trữ tệp tạm thời, tăng tốc độ truy cập dữ liệu từ AWS S3.

## 3.3. Network cache:

### 3.3.1. Khái niệm:

- Network Cache là bộ nhớ đệm lưu trữ dữ liệu yêu cầu mạng từ trước đó.
- Network Cache thường nằm ở **tầng 7 (Tầng ứng dụng)** trong mô hình OSI. Cụ thể:
- **Browser cache** và **CDN cache** hoạt động ở tầng ứng dụng (tầng 7), nơi lưu trữ các tài nguyên HTTP/HTTPS như HTML, hình ảnh, hoặc tệp JS.

- Server cache (như Redis, Memcached) cũng thuộc tầng ứng dụng, lưu trữ kết quả truy vấn hoặc dữ liệu đã xử lý.

### 3.3.2. Phân loại:

- DNS Cache: Trình duyệt lưu trữ tạm các DNS records vào trong các thiết bị, trình duyệt.
- Proxy Cache: Máy chủ proxy lưu trữ dữ liệu để phục vụ nhiều người dùng (ví dụ: trong mạng công ty).
- CDN Cache: Mạng phân phối nội dung (Content Delivery Network) lưu trữ dữ liệu trên các máy chủ cạnh (edge servers) gần người dùng.

### 3.3.3. Chi tiết:

#### 3.3.3.1. DNS và DNS Cache:

##### 3.3.3.1.1. Khái niệm:

- Hệ thống Tên miền (**Domain Name System - DNS**) hoạt động như một “danh bạ Internet”, chuyển đổi tên miền dễ nhớ (ví dụ: [www.google.com](http://www.google.com)) thành địa chỉ IP (ví dụ: 172.217.167.78) để máy tính kết nối với máy chủ. DNS giúp người dùng truy cập các trang web mà không cần ghi nhớ các dãy số phức tạp.

- DNS bao gồm các thành phần chính:

- **Tên miền (Domain Name):** Địa chỉ thân thiện với người dùng, như [example.com](http://example.com).
- **Máy chủ DNS (DNS Server):** Lưu trữ và xử lý thông tin chuyển đổi tên miền thành IP.
- **Resolver:** Phần mềm trên thiết bị người dùng, gửi yêu cầu tra cứu đến máy chủ DNS.
- **Bản ghi DNS (DNS Record):** Dữ liệu liên kết tên miền với IP hoặc các dịch vụ khác.

- DNS Cache là bộ nhớ đệm lưu trữ thông tin ánh xạ tên miền và địa chỉ IP tạm thời trên thiết bị người dùng, trình duyệt, hệ điều hành hoặc máy chủ DNS để tăng tốc độ truy vấn DNS. Thay vì thực hiện truy vấn mới, hệ thống kiểm tra cache trước, giúp:

- Tăng tốc độ truy cập: Giảm thời gian tải trang web.
- Giảm tải cho máy chủ DNS: Hạn chế số lượng yêu cầu gửi đến máy chủ.
- Tiết kiệm băng thông: Đặc biệt hữu ích cho các mạng lớn.

- DNS không chỉ tồn tại ở một nơi mà được lưu trữ ở nhiều cấp độ khác nhau, mỗi cấp có vai trò riêng:

- **Trình duyệt (Browser Cache):** Các trình duyệt như Chrome, Firefox lưu trữ bản ghi DNS để tăng tốc độ tải trang. Cache này thường có TTL ngắn và bị xóa khi đóng trình duyệt hoặc xóa dữ liệu duyệt web.
  - **Hệ điều hành (OS Cache):** Hệ điều hành (Windows, macOS, Linux) duy trì một bộ nhớ đệm DNS riêng, được sử dụng bởi tất cả các ứng dụng trên thiết bị. Ví dụ, Windows lưu trữ trong DNS Client Service.
  - **Router Cache:** Nhiều bộ định tuyến (router) lưu trữ DNS Cache để phục vụ các thiết bị trong mạng nội bộ, giảm truy vấn đến máy chủ DNS bên ngoài.
  - **ISP hoặc DNS Resolver Cache:** Nhà cung cấp dịch vụ Internet (ISP) hoặc máy chủ DNS công cộng (như Google DNS, Cloudflare) lưu trữ cache để phục vụ nhiều người dùng, giúp giảm tải cho các máy chủ DNS cấp cao hơn (Root, TLD, Authoritative).
  - **Máy chủ DNS nội bộ (Local DNS Server):** Trong các tổ chức lớn, máy chủ DNS nội bộ lưu trữ cache để xử lý truy vấn cho mạng nội bộ.
- Mỗi bản ghi DNS trong cache có TTL (Time to Live), xác định thời gian bản ghi được lưu trước khi hết hạn. Ví dụ, nếu TTL là 3600 giây (1 giờ), cache sẽ được sử dụng trong thời gian này, sau đó hệ thống phải tra cứu lại.

#### 3.3.3.1.2. Cách hoạt động:

- Khi người dùng truy cập một tên miền (ví dụ: www.example.com), hệ thống thực hiện các bước sau:
  - Kiểm tra cache trình duyệt:
    - ▶ Trình duyệt (Chrome, Firefox, v.v.) kiểm tra bộ nhớ đệm DNS nội bộ.
    - ▶ Nếu tìm thấy bản ghi hợp lệ (chưa hết hạn TTL - Time to Live), trình duyệt sử dụng địa chỉ IP ngay lập tức.
    - ▶ Ví dụ: Chrome lưu cache DNS trong tối đa 60 giây, tùy thuộc vào TTL.
  - Kiểm tra cache hệ điều hành:
  - Nếu trình duyệt không có bản ghi, yêu cầu được chuyển đến cache DNS của hệ điều hành (Windows, macOS, Linux).
  - Ví dụ: Windows sử dụng DNS Client Service để lưu cache, có thể xem bằng **lệnh ipconfig /displaydns**.
  - Nếu bản ghi tồn tại và chưa hết TTL, địa chỉ IP được trả về.
  - Kiểm tra cache router:

- ▶ Nếu hệ điều hành không có bản ghi, yêu cầu được gửi đến router (nếu có cache DNS).
- ▶ Router lưu cache để phục vụ nhiều thiết bị trong mạng nội bộ, giảm truy vấn ra ngoài.
- Kiểm tra cache DNS Resolver:
  - ▶ Nếu router không có bản ghi, yêu cầu được gửi đến DNS Resolver (thường do ISP hoặc nhà cung cấp như Google DNS - 8.8.8.8 cung cấp).
    - Resolver kiểm tra cache của nó. Nếu có bản ghi, địa chỉ IP được trả về và lưu vào cache của router, hệ điều hành, và trình duyệt (nếu được cấu hình).
    - Phân giải DNS đầy đủ (nếu không có cache):
    - Nếu resolver không có bản ghi, nó thực hiện truy vấn đệ quy qua:
      - Root Servers: Xác định TLD (như .com).
      - TLD Servers: Chuyển hướng đến máy chủ DNS có thẩm quyền.
      - Authoritative Servers: Cung cấp địa chỉ IP cuối cùng.
- Kết quả được lưu vào cache ở tất cả các cấp (resolver, router, hệ điều hành, trình duyệt) với TTL do máy chủ DNS chỉ định.

### **3.3.3.1.3. Tác dụng của DNS cache:**

#### **• Tăng Tốc Độ Truy Cập Website:**

- DNS Cache lưu trữ ánh xạ tên miền (như google.com) sang địa chỉ IP (như 172.217.167.78). Các truy vấn sau sử dụng cache thay vì gửi yêu cầu đến máy chủ DNS, giảm thời gian tải trang từ vài trăm mili giây xuống gần như tức thì.
- **Ví dụ:** Truy cập facebook.com lần thứ hai trong ngày chỉ mất 10ms nhờ cache, so với 200ms nếu phân giải đầy đủ.

#### **• Giảm Tải cho Máy Chủ DNS:**

- Cache giảm số lượng truy vấn gửi đến máy chủ DNS (Root, TLD, Authoritative), giúp máy chủ hoạt động hiệu quả và ít bị quá tải trong giờ cao điểm.
- **Ví dụ:** Một tổ chức với 1.000 nhân viên truy cập office.com có thể giảm hàng nghìn truy vấn DNS nhờ cache nội bộ.

#### **• Tiết Kiệm Băng Thông và Tối Ưu Hóa Mạng:**

- DNS Cache tại ISP hoặc router giảm lưu lượng mạng bằng cách phục vụ truy vấn từ bản ghi lưu sẵn, đặc biệt hữu ích cho mạng lớn hoặc khu vực băng thông hạn chế.

- **Ví dụ:** ISP sử dụng cache để giảm 30% lưu lượng DNS khi người dùng truy cập các trang phổ biến như YouTube.
- **Cải Thiện Trải Nghiệm Người Dùng:**
  - Thời gian phản hồi nhanh hơn mang lại trải nghiệm duyệt web mượt mà, đặc biệt trên thiết bị di động hoặc trong ứng dụng thời gian thực như trò chơi trực tuyến.
  - **Ví dụ:** Trong game online, cache DNS giúp giảm độ trễ khi kết nối đến máy chủ.
- **Hỗ Trợ Khi Mất Kết Nối:**
  - Nếu máy chủ DNS không khả dụng, cache cục bộ vẫn cung cấp bản ghi DNS, đảm bảo truy cập các website quen thuộc.
  - **Ví dụ:** Trong sự cố mất kết nối ISP, người dùng vẫn truy cập được cloudflare.com nếu IP đã lưu trong cache.
- **Tăng Hiệu Quả cho IoT và CDN:**
  - DNS Cache tối ưu hóa truy vấn trong mạng IoT với hàng triệu thiết bị và hỗ trợ CDN (như Cloudflare) định tuyến người dùng đến máy chủ gần nhất.
  - **Ví dụ:** Cloudflare sử dụng cache để giảm thời gian tải trang từ 150ms xuống 20ms.

#### 3.3.3.1.4. Các DNS phổ biến và công ty vận hành

##### 3.3.3.1.4.1. Google Public DNS

- **Nhà cung cấp:** Google LLC
- **Địa chỉ IP:** 8.8.8.8, 8.8.4.4 (IPv6: 2001:4860:4860::8888, 2001:4860:4860::8844)
- **Ưu điểm:**
  - Tốc độ cao nhờ mạng lưới máy chủ toàn cầu của Google, giảm độ trễ trung bình xuống dưới 20ms.
  - Hỗ trợ DNSSEC để xác thực bản ghi DNS, tăng cường bảo mật.
  - Miễn phí, dễ cấu hình trên mọi thiết bị (Windows, macOS, router).
  - Tích hợp tốt với các dịch vụ Google (như YouTube, Google Drive).
  - Hỗ trợ DoH và DoT (thử nghiệm từ 2023), mã hóa truy vấn DNS.
- **Nhược điểm:**
  - Một số tính năng nâng cao (như lọc nội dung, phân tích truy vấn) chỉ có trong Google Workspace.
  - Quyền riêng tư bị nghi ngại do Google thu thập dữ liệu truy vấn (mặc dù được tuyên bố ẩn danh).
- **Ứng dụng thực tế:**
  - Phù hợp cho người dùng cá nhân muốn tốc độ cao khi truy cập các trang như googleusercontent.com/youtube.com/0.
  - Doanh nghiệp nhỏ sử dụng Google Workspace để quản lý DNS nội bộ.

#### 3.3.3.1.4.2. Cloudflare DNS

- **Nhà cung cấp:** Cloudflare, Inc.
- **Địa chỉ IP:** 1.1.1.1, 1.0.0.1 (IPv6: 2606:4700:4700::1111, 2606:4700:4700::1001)
- **Ưu điểm:**
  - Tốc độ nhanh nhất trong các DNS công cộng (theo DNSPerf, trung bình 11ms).
  - Hỗ trợ DoH và DoT, mã hóa truy vấn để bảo vệ quyền riêng tư.
  - Miễn phí, cung cấp tính năng chặn quảng cáo và phần mềm độc hại qua 1.1.1.1 for Families.
  - Tích hợp với CDN của Cloudflare, tối ưu hóa truy cập các website lớn như netflix.com
  - Chính sách không lưu trữ dữ liệu truy vấn lâu dài, tăng cường quyền riêng tư.
- **Nhược điểm:**
  - Tính năng nâng cao như phân tích lưu lượng DNS hoặc lọc tùy chỉnh chỉ có trong gói Cloudflare Pro hoặc Gateway.
  - Không hỗ trợ đầy đủ DNSSEC (chỉ hỗ trợ một phần).
- **Ứng dụng thực tế:**
  - Người dùng cá nhân muốn bảo mật và chặn quảng cáo khi duyệt web.
  - Doanh nghiệp sử dụng Cloudflare CDN để tăng tốc website.

#### 3.3.3.1.4.3. Quad9 DNS

- **Nhà cung cấp:** Quad9 Foundation (hỗ trợ bởi IBM)
- **Địa chỉ IP:** 9.9.9.9, 149.112.112.112 (IPv6: 2620:fe::fe, 2620:fe::9)
- **Ưu điểm:**
  - Tập trung vào bảo mật, tự động chặn các tên miền độc hại dựa trên dữ liệu từ IBM X-Force.
  - Hỗ trợ DNSSEC, DoH, và DoT, đảm bảo an toàn và quyền riêng tư.
  - Miễn phí, không thu thập dữ liệu cá nhân, phù hợp cho người dùng nhạy cảm về quyền riêng tư.
  - Độ tin cậy cao nhờ mạng lưới máy chủ toàn cầu.
- **Nhược điểm:**
  - Tốc độ chậm hơn một chút so với Cloudflare hoặc Google (trung bình 15-20ms).
  - Thiếu các tính năng nâng cao như lọc nội dung tùy chỉnh hoặc phân tích lưu lượng.
- **Ứng dụng thực tế:**
  - Cá nhân hoặc tổ chức muốn bảo vệ khỏi phần mềm độc hại khi truy cập các trang như malicious-site.com.
  - Trường học hoặc thư viện cần DNS an toàn mà không yêu cầu cấu hình phức tạp.

#### 3.3.3.1.4.4. OpenDNS

- **Nhà cung cấp:** Cisco Systems, Inc.
- **Địa chỉ IP:** 208.67.222.222, 208.67.220.220 (IPv6: 2620:0:ccc::2, 2620:0:ccd::2)

- **Ưu điểm:**
  - Gói trả phí (Cisco Umbrella) cung cấp lọc nội dung tùy chỉnh, phân tích lưu lượng, và bảo vệ nâng cao.
  - Hỗ trợ DNSSEC và tích hợp tốt với các giải pháp bảo mật doanh nghiệp.
- **Nhược điểm:**
  - Tính năng nâng cao (lọc chi tiết, báo cáo) chỉ có trong gói trả phí.
  - Không hỗ trợ DoH hoặc DoT, làm giảm bảo mật so với Cloudflare hoặc Quad9.
- **Ứng dụng thực tế:**
  - **Gia đình muốn chặn nội dung không phù hợp cho trẻ em.**
  - **Doanh nghiệp sử dụng Cisco Umbrella để quản lý truy cập mạng nội bộ.**

#### 3.3.3.1.4.5. Neustar DNS

- **Nhà cung cấp:** Neustar, Inc. (nay thuộc TransUnion)
- **Địa chỉ IP:** 156.154.70.1, 156.154.71.1 (IPv6: 2610:a1:1018::1, 2610:a1:1019::1)
- **Ưu điểm:**
  - Cung cấp cả gói miễn phí (UltraDNS Public) và trả phí với tốc độ cao, độ tin cậy tốt.
  - Tính năng nâng cao như chặn phần mềm độc hại, lọc nội dung, và phân tích lưu lượng trong gói trả phí.
  - Hỗ trợ DNSSEC và tích hợp với các giải pháp bảo mật doanh nghiệp.
  - Phù hợp cho các tổ chức cần DNS ổn định và an toàn.
- **Nhược điểm:**
  - Gói miễn phí hạn chế tính năng (chỉ cung cấp phân giải DNS cơ bản).
  - Không hỗ trợ DoH hoặc DoT, kém hơn về bảo mật mã hóa.
  - Tốc độ không nhanh bằng Cloudflare hoặc Google (trung bình 20ms).
- **Ứng dụng thực tế:**
  - Doanh nghiệp cần DNS đáng tin cậy cho các ứng dụng nội bộ.
  - Tổ chức tài chính sử dụng gói trả phí để bảo vệ khỏi tấn công DNS.

#### 3.3.3.1.5. Rủi ro của DNS Cache:

- **DNS Cache Poisoning (Ô nhiễm bộ nhớ đệm):**
- **Mô tả:** Kẻ tấn công chèn bản ghi DNS giả mạo vào cache, khiến người dùng bị chuyển hướng đến các trang web độc hại hoặc lừa đảo.
- **Ví dụ:** Một hacker giả mạo bản ghi của bank.com trong cache router, dẫn người dùng đến trang giả mạo để đánh cắp thông tin đăng nhập.
- **Tác động:** Gây rủi ro bảo mật nghiêm trọng, bao gồm đánh cắp dữ liệu, cài đặt phần mềm độc hại, hoặc lừa đảo tài chính.
- **Yếu tố kỹ thuật:** Xảy ra khi DNS không sử dụng giao thức bảo mật như DNSSEC hoặc DoH/DoT, đặc biệt trong các mạng công cộng.



- **Thông Tin Lỗi Thời (Stale Data):**

- **Mô tả:** Cache lưu giữ bản ghi DNS cũ sau khi địa chỉ IP của tên miền thay đổi, dẫn đến truy cập không chính xác hoặc thất bại.
- **Ví dụ:** Một website như example.com chuyển sang IP mới, nhưng cache trình duyệt vẫn lưu IP cũ trong 24 giờ (do TTL chưa hết hạn), gây lỗi “không thể kết nối”.
- **Tác động:** Làm gián đoạn trải nghiệm người dùng, đặc biệt trong các hệ thống yêu cầu tính cập nhật cao như CDN hoặc dịch vụ cân bằng tải.
- **Yếu tố kỹ thuật:** Phụ thuộc vào giá trị TTL; TTL dài (như 24 giờ) làm tăng nguy cơ lỗi thời.

### 3.3.3.2. CDNcache:

#### 3.3.3.2.1. Khái niệm:

CDN cache (Content Delivery Network cache) là bộ nhớ đệm được sử dụng bởi các mạng phân phối nội dung (CDN) để lưu trữ tạm thời các tài nguyên web (như hình ảnh, video, tệp HTML, CSS, JavaScript) tại các máy chủ biên (edge servers) gần người dùng nhất. Mục tiêu là tăng tốc độ tải trang web và giảm tải cho máy chủ gốc (origin server).

#### 3.3.3.2.2. Cách hoạt động:

##### 3.3.3.2.2.1. Lưu trữ nội dung tại máy chủ biên:

- Khi một website tích hợp CDN, các nội dung tĩnh (như hình ảnh, video, CSS, JavaScript) hoặc thậm chí nội dung động (nếu được cấu hình) sẽ được sao chép từ máy chủ gốc đến các máy chủ biên của CDN.
- Quá trình này thường được kích hoạt khi có yêu cầu đầu tiên từ người dùng hoặc được cấu hình trước bởi quản trị viên website.

##### 3.3.3.2.2.2. Xử lý yêu cầu của người dùng:

- Khi người dùng truy cập website (ví dụ: mở một trang web hoặc xem video), yêu cầu của họ được gửi đến hệ thống DNS.
- CDN sử dụng Anycast DNS hoặc các thuật toán định tuyến địa lý (geo-routing) để chuyển hướng yêu cầu đến máy chủ biên gần nhất về mặt địa lý hoặc có độ trễ thấp nhất.

##### 3.3.3.2.2.3. Kiểm tra cache tại máy chủ biên:

- Máy chủ biên kiểm tra xem nội dung yêu cầu (ví dụ: một tệp hình ảnh) có trong bộ nhớ đệm (cache) hay không:

- ▶ Cache hit: Nếu nội dung đã có trong cache và chưa hết hạn (dựa trên thời gian sống - TTL), máy chủ biên trả nội dung trực tiếp cho người dùng. Điều này giúp giảm thời gian tải đáng kể.
- ▶ Cache miss: Nếu nội dung không có trong cache hoặc đã hết hạn, máy chủ biên sẽ gửi yêu cầu đến máy chủ gốc để lấy nội dung mới.

#### **3.3.3.2.2.4. Lấy nội dung từ máy chủ gốc (nếu cần):**

- Trong trường hợp cache miss, máy chủ biên liên lạc với máy chủ gốc để tải nội dung yêu cầu.
- Sau khi nhận được nội dung, máy chủ biên lưu trữ bản sao vào cache (theo cấu hình TTL) và gửi nội dung về cho người dùng.
- Quá trình này chỉ xảy ra lần đầu hoặc khi nội dung trong cache cần được làm mới.

#### **3.3.3.2.2.5. Phân phối nội dung cho người dùng:**

- Máy chủ biên gửi nội dung từ cache hoặc từ máy chủ gốc (nếu vừa lấy) đến người dùng qua kết nối mạng tối ưu, thường sử dụng các giao thức như HTTP/2 hoặc QUIC để tăng tốc độ.

#### **3.3.3.2.2.6. Quản lý và cập nhật cache:**

- Thời gian sống (TTL): Mỗi tệp trong cache có thời gian tồn tại được cấu hình (ví dụ: 1 giờ, 1 ngày). Khi TTL hết, nội dung sẽ bị xóa hoặc làm mới từ máy chủ gốc.
- Xóa cache (Purge): Quản trị viên có thể chủ động xóa cache trên CDN khi nội dung trên máy chủ gốc được cập nhật (ví dụ: thay đổi hình ảnh hoặc mã CSS).
- Tùy chỉnh cache: CDN cho phép cấu hình để quyết định nội dung nào được lưu trữ, thời gian lưu trữ, hoặc cách xử lý các yêu cầu động (dynamic content).

#### **3.3.3.2.3. Ứng dụng và lợi ích:**

Lợi ích của CDN Cache (bộ nhớ đệm trong Content Delivery Network) bao gồm các khía cạnh sau, giúp tối ưu hóa hiệu suất website, cải thiện trải nghiệm người dùng và giảm tải hệ thống:

##### **3.3.3.2.3.1. Tăng tốc độ tải trang:**

- CDN cache lưu trữ nội dung tĩnh (như hình ảnh, video, CSS, JavaScript) tại các máy chủ biên gần người dùng, giảm độ trễ (latency) khi truyền dữ liệu.
- Người dùng nhận được nội dung nhanh hơn, đặc biệt khi truy cập từ các khu vực xa máy chủ gốc (ví dụ: truy cập website Mỹ từ Việt Nam).

##### **3.3.3.2.3.2. Giảm tải cho máy chủ gốc:**

- Các yêu cầu được xử lý trực tiếp từ cache tại máy chủ biên, giảm số lượng truy vấn gửi đến máy chủ gốc.

- Tiết kiệm tài nguyên CPU, băng thông và chi phí vận hành cho máy chủ chính, đặc biệt trong các đợt lưu lượng truy cập cao (như sự kiện livestream hoặc flash sale).

#### **3.3.3.2.3.3. Cải thiện độ tin cậy và tính sẵn sàng:**

- Nếu máy chủ gốc gặp sự cố (downtime), nội dung trong cache tại máy chủ biên vẫn có thể được phân phối, đảm bảo website hoạt động liên tục.
- CDN giúp phân phối lưu lượng, tránh tình trạng quá tải máy chủ.

#### **3.3.3.2.3.4. Tối ưu hóa trải nghiệm người dùng:**

- Tốc độ tải trang nhanh hơn dẫn đến trải nghiệm mượt mà, tăng sự hài lòng và giữ chân người dùng lâu hơn.
- Hỗ trợ tốt cho các ứng dụng nặng như streaming video, game trực tuyến hoặc website thương mại điện tử.

#### **3.3.3.2.3.5. Tăng khả năng mở rộng (scalability):**

- CDN cache giúp website xử lý lưu lượng truy cập lớn mà không cần nâng cấp phần cứng máy chủ gốc.
- Phù hợp với các sự kiện có lượng truy cập đột biến, như ra mắt sản phẩm hoặc chương trình khuyến mãi.

#### **3.3.3.2.3.6. Cải thiện SEO:**

- Tốc độ tải trang là yếu tố quan trọng trong xếp hạng công cụ tìm kiếm (như Google). CDN cache giúp cải thiện chỉ số này, tăng khả năng hiển thị website.
- Giảm tỷ lệ thoát trang (bounce rate) do thời gian tải nhanh hơn.

#### **3.3.3.2.3.7. Tiết kiệm chi phí băng thông:**

- Bằng cách phục vụ nội dung từ cache, CDN giảm lượng dữ liệu truyền từ máy chủ gốc, giúp tiết kiệm chi phí băng thông, đặc biệt với các website có lưu lượng lớn.

#### **3.3.3.2.3.8. Tăng cường bảo mật:**

- Nhiều CDN cung cấp tính năng bảo mật bổ sung như chống tấn công DDoS, mã hóa SSL/TLS, và tường lửa ứng dụng web (WAF), bảo vệ website khỏi các mối đe dọa.
- Cache giúp giảm tiếp xúc trực tiếp của máy chủ gốc với các yêu cầu độc hại.

#### **3.3.3.2.3.9. Hỗ trợ phân phối nội dung toàn cầu:**

- Với mạng lưới máy chủ biên phân bố khắp thế giới, CDN cache đảm bảo người dùng từ mọi khu vực nhận được nội dung nhanh chóng và ổn định.
- Hỗ trợ các website quốc tế hoặc ứng dụng có người dùng toàn cầu (như Netflix, YouTube).

### 3.3.3.2.3.10. Linh hoạt trong quản lý nội dung:

- Quản trị viên có thể tùy chỉnh thời gian sống (TTL) của cache, xóa cache (purge) khi cần cập nhật nội dung, hoặc cấu hình để xử lý cả nội dung động.
- Một số CDN hiện đại tích hợp edge computing để xử lý logic ngay tại máy chủ biên, tăng hiệu quả.

## 3.4. Software cache

### 3.4.1. Các loại cache của hệ điều hành:

- **Page Cache (Disk Cache):**

- **Khái niệm:** Dữ liệu từ ổ đĩa được lưu trữ tạm trong bộ nhớ RAM để truy cập nhanh hơn.
- **Cách hoạt động:** Khi một chương trình truy cập tệp, hệ điều hành sẽ lưu bản sao vào RAM; nếu truy cập lại, sẽ dùng bản sao trong RAM thay vì đọc lại từ đĩa.
- **Ứng dụng:** Tăng tốc truy cập file/tài liệu.
- **Lợi ích:** Giảm I/O disk, tăng hiệu suất hệ thống.

**Ví dụ:** Bạn mở file PDF dung lượng 50MB từ ổ cứng lần đầu, mất 3 giây. Khi đóng rồi mở lại file này lần nữa, chỉ mất 0.2 giây — vì nội dung đã được cache trong RAM.

- **Buffer Cache**

- **Khái niệm:** Dữ liệu buffer cho các thao tác ghi/đọc đĩa.
- **Cách hoạt động:** Ghi dữ liệu vào bộ đệm (RAM) trước khi ghi thực sự xuống ổ đĩa.
- **Ứng dụng:** Tăng hiệu suất ghi đĩa.
- **Lợi ích:** Giảm thời gian chờ ghi/đọc, gom nhóm các thao tác I/O.

**Ví dụ:** Khi bạn copy một thư mục lớn (ví dụ 10GB), hệ thống sẽ ghi dữ liệu vào RAM trước rồi sau đó ghi dần xuống ổ đĩa. Lệnh cp hoàn thành nhanh nhưng đèn ổ cứng vẫn nhấp nháy vài phút.

- **Dentry Cache (Directory Entry Cache)**

- **Khái niệm:** Cache thông tin về các đường dẫn thư mục/tập tin (tên, inode).
- **Cách hoạt động:** Lưu thông tin về cấu trúc cây thư mục đã từng được truy cập.
- **Ứng dụng:** Duyệt file/folder.
- **Lợi ích:** Tăng tốc tìm kiếm tệp.

**Ví dụ:** Khi bạn dùng lệnh `ls /home/user/projects` lần đầu, nó mất 0.5 giây; lần thứ hai gần như tức thì. Vì thông tin tên file và đường dẫn đã được lưu trong dentry cache.

### Điểm riêng của hệ Linux:

- **Transparent Page Cache:** Linux tự động sử dụng RAM còn trống làm cache mà không yêu cầu ứng dụng can thiệp.
- **Command kiểm tra cache:**
  - free -h: xem “cached” memory
  - cat /proc/meminfo: thông tin chi tiết hơn.
- **Drop cache:** có thể xóa cache thủ công bằng lệnh `echo 3 > /proc/sys/vm/drop_caches`.
- **Slab Allocator:** quản lý các loại cache như dentry, inode theo cách tối ưu hóa bộ nhớ nhỏ lẻ.

### Điểm riêng của hệ Windows:

- **System File Cache:** Windows sử dụng một vùng RAM gọi là System Working Set để cache file.
- **SuperFetch/Prefetch:**
  - **Prefetch:** Ghi nhớ các file thường được sử dụng khi khởi động để load nhanh hơn.
  - **SuperFetch (trên Windows Vista trở đi):** học thói quen sử dụng ứng dụng để preload vào RAM.
- **RamMap:** công cụ để xem chi tiết cache RAM trên Windows.
- **Tuning hạn chế:** Ít quyền kiểm soát trực tiếp cache hơn Linux.

### 3.4.2. Web cache:

#### 3.4.2.1. Cache của trình duyệt web (browser cache):

##### 1. Browser cache là gì?

C là một cơ chế lưu trữ tạm thời được tích hợp trong trình duyệt web, giúp lưu trữ các tài nguyên tải về từ các trang web để tăng tốc độ truy cập và giảm tải cho máy chủ.

Khi người dùng sử dụng trình duyệt trên máy tính để truy cập vào một trang web bất kỳ, trình duyệt sẽ tải các thành phần dữ liệu từ trên web server xuống như hình ảnh, font chữ, file text... và kết hợp các thành phần này thành giao diện trang web hoàn chỉnh hiển thị trên trình duyệt. Các thành phần dữ liệu đó sẽ được lưu tạm thời vào bộ nhớ đệm trên trình duyệt. Và đó là bộ nhớ đệm (cache).

Browser cache (bộ nhớ đệm của trình duyệt) là một vùng lưu trữ dữ liệu trên ổ cứng hoặc trên RAM của máy tính, dùng để lưu trữ tạm thời dữ liệu được trình duyệt tải về. Các dữ liệu sẽ được lưu tại bộ nhớ đệm này thường là các file hình ảnh, file html, file

css, file javascript, tùy thuộc vào webserver chỉ định sẽ lưu loại nào trên cache. Và các dữ liệu tạm thời này sẽ tồn tại trong một thời gian ngắn được chỉ định, sau thời gian chỉ định thì dữ liệu đó sẽ được xóa hoặc làm mới, cập nhật lại.

## **2) Vai trò của Browser cache:**

Browser cache đóng vai trò quan trọng trong tối ưu hóa hiệu suất và tăng tốc độ tải trang web ở phía client (front-end). Khi người dùng truy cập một trang web, trình duyệt lưu trữ các tài nguyên của trang đó trong bộ nhớ cache. Khi trang web được tải lại hoặc người dùng truy cập vào các trang khác trên cùng một trình duyệt, các tài nguyên này có thể được sử dụng lại từ cache thay vì phải tải lại từ máy chủ. Điều này mang lại nhiều lợi ích quan trọng trong việc cải thiện trải nghiệm người dùng và giảm tải cho máy chủ.

### **• Tăng tốc độ truy cập trang web:**

Đây là vai trò quan trọng nhất của Browser cache trong Front-end. Nhờ việc lưu trữ tạm thời các tài nguyên tĩnh như hình ảnh, CSS, JavaScript, ... vào bộ nhớ đệm, trình duyệt không cần tải xuống lại những tài nguyên này từ máy chủ web mỗi khi người dùng truy cập trang. Điều này giúp giảm đáng kể thời gian tải trang, đặc biệt là đối với những trang web có nhiều nội dung tĩnh.

Khi trình duyệt gặp lại cùng một tài nguyên trong quá trình duyệt web, nó sẽ kiểm tra xem tài nguyên này đã được lưu trữ trong cache hay chưa. Nếu tài nguyên tồn tại trong cache và chưa hết hạn, trình duyệt sẽ sử dụng lại tài nguyên từ cache thay vì phải tải lại từ máy chủ. Điều này giúp giảm thời gian tải trang đáng kể, vì các tài nguyên chỉ cần được truy cập từ bộ nhớ cache trên máy tính của người dùng thay vì từ xa qua mạng Internet.

Bởi vì tải các tài nguyên từ cache nhanh hơn nhiều so với tải từ máy chủ, Browser cache giúp giảm thời gian tải trang tổng thể. Điều này có tác động đáng kể đến trải nghiệm người dùng, đặc biệt là trên các trang web có nhiều tài nguyên lớn như hình ảnh chất lượng cao hoặc video.

### **• Giảm tải cho máy chủ web:**

Khi người dùng truy cập trang web, máy chủ web phải xử lý yêu cầu và gửi dữ liệu cho người dùng. Việc sử dụng Browser cache giúp giảm số lượng yêu cầu gửi đến máy chủ web, từ đó giảm tải cho máy chủ và giúp máy chủ hoạt động hiệu quả hơn.

Bởi vì khi trình duyệt sử dụng lại các tài nguyên từ cache thì sẽ không có yêu cầu mới nào được gửi đến máy chủ. Điều này giảm lượng yêu cầu mà máy chủ phải xử lý. Thay vì phải gửi lại các tài nguyên cho mỗi yêu cầu, máy chủ chỉ cần xử lý

yêu cầu ban đầu và sau đó trình duyệt sẽ chia sẻ bản sao đó với các trình duyệt khác. Nhờ vậy, máy chủ web chỉ cần xử lý một yêu cầu thay vì 1000 yêu cầu, giúp giảm tải đáng kể cho máy chủ.

- **Tiết kiệm băng thông:**

Vai trò của Browser cache trong việc tiết kiệm băng thông liên quan đến việc giảm lượng dữ liệu phải chuyển qua mạng giữa trình duyệt và máy chủ.

Khi trình duyệt sử dụng lại các tài nguyên từ cache, không cần phải tải lại các tài nguyên đó từ máy chủ. Điều này giảm lượng dữ liệu phải chuyển qua mạng, giúp tiết kiệm băng thông. Thay vì gửi lại toàn bộ tài nguyên cho mỗi yêu cầu, trình duyệt chỉ cần gửi một yêu cầu nhỏ để kiểm tra xem tài nguyên trong cache còn hợp lệ hay không. Nếu tài nguyên hợp lệ, nó sẽ được sử dụng lại từ cache mà không cần tải lại từ máy chủ. Điều này giúp giảm lượng dữ liệu truyền qua mạng và giảm sử dụng băng thông.

Việc tiết kiệm băng thông mạng có nhiều lợi ích. Trước tiên, nó giúp cải thiện trải nghiệm người dùng, đặc biệt là trên các kết nối mạng chậm hoặc có giới hạn băng thông. Bởi vì tài nguyên đã được lưu trữ trong cache, trình duyệt không cần phải tải lại chúng từ máy chủ, giúp giảm thời gian tải trang và tăng tốc độ duyệt web.

Thứ hai, việc giảm sử dụng băng thông cũng có lợi cho máy chủ web. Khi các tài nguyên đã được lưu trữ trong cache, máy chủ không cần phải gửi lại các tài nguyên đó cho mỗi yêu cầu. Điều này giảm tải cho máy chủ và giúp nó xử lý được nhiều yêu cầu từ các người dùng khác nhau mà không ảnh hưởng nhiều đến hiệu suất và thời gian phản hồi.

Tóm lại, Browser cache giúp tiết kiệm băng thông bằng cách giảm lượng dữ liệu phải chuyển qua mạng. Sử dụng lại các tài nguyên từ cache giúp trình duyệt không cần phải tải lại từ máy chủ, giảm tải lượng dữ liệu và giảm sử dụng băng thông. Điều này cải thiện trải nghiệm người dùng và giảm tải cho máy chủ web, cho phép nó xử lý được nhiều yêu cầu từ các người dùng khác nhau một cách hiệu quả.

- **Hỗ trợ hoạt động ngoại tuyến:**

Một số trình duyệt cho phép người dùng truy cập các trang web đã truy cập trước đó ngay cả khi không có kết nối internet. Điều này nhờ vào việc trình duyệt lưu trữ các trang web vào bộ nhớ đệm, cho phép người dùng truy cập nội dung đã tải xuống trước đó.

Khi người dùng truy cập vào một trang web mà các tài nguyên của trang đã được lưu trữ trong cache, trình duyệt có thể sử dụng lại chúng mà không cần phụ thuộc vào kết nối mạng. Điều này có ý nghĩa đặc biệt khi người dùng không có kết nối mạng hoặc kết nối mạng bị gián đoạn. Thay vì hiển thị thông báo lỗi hoặc trang trắng, trình duyệt

có thể tải các tài nguyên đã lưu trữ trong cache và hiển thị nội dung trang web cho người dùng.

Vai trò hoạt động ngoại tuyến của Browser cache có nhiều ứng dụng. Ví dụ, trong các ứng dụng web dựa trên HTML5, nó cho phép các ứng dụng hoạt động ngoại tuyến và lưu trữ dữ liệu trong cache để truy cập khi không có kết nối mạng. Điều này hữu ích cho các ứng dụng di động như ứng dụng đọc tin tức, ứng dụng ghi chú hoặc ứng dụng xem tài liệu, cho phép người dùng tiếp tục sử dụng các tính năng của ứng dụng mà không phụ thuộc hoàn toàn vào kết nối mạng.

Tóm lại, vai trò hoạt động ngoại tuyến của Browser cache đóng vai trò quan trọng trong việc cải thiện trải nghiệm người dùng khi truy cập vào các trang web trong tình huống không có kết nối mạng hoặc kết nối mạng không ổn định. Bằng cách lưu trữ các tài nguyên của trang web trong cache, trình duyệt có thể sử dụng lại chúng để hiển thị nội dung trang web mà không cần phụ thuộc vào kết nối mạng, đảm bảo rằng người dùng vẫn có thể truy cập vào thông tin và chức năng của trang web một cách liên tục và liền mạch.

### 3) Các loại bộ nhớ đệm (Browser Cache):

Bộ nhớ đệm trình duyệt có 2 loại:

- **Private cache:**

Private cache (bộ nhớ đệm riêng) dành riêng cho một người dùng. Bạn có thể đã thấy “bộ nhớ đệm” trong cài đặt trình duyệt của mình. Bộ nhớ cache của trình duyệt lưu giữ tất cả các tài liệu được người dùng tải xuống qua HTTP. Bộ nhớ đệm này được sử dụng để cung cấp các tài liệu đã truy cập để điều hướng lùi / chuyển tiếp, lưu, xem dưới dạng nguồn, ... mà không yêu cầu thêm một chuyến đi tới máy chủ. Nó cũng cải thiện khả năng duyệt ngoại tuyến các nội dung được lưu trong bộ nhớ cache.

- **Public cache (Shared cache):**

Shared cache (bộ nhớ đệm chia sẻ) là một nơi lưu trữ tạm thời cho các dữ liệu được truy cập thường xuyên. Nó được đặt giữa máy chủ web và cơ sở dữ liệu và được sử dụng để lưu trữ các dữ liệu đã truy cập thường xuyên để giảm số lượng yêu cầu gửi đến cơ sở dữ liệu. Điều này giúp cải thiện hiệu suất của ứng dụng web bằng cách giảm số lượng yêu cầu gửi đến cơ sở dữ liệu.

*Ví dụ: một công ty của bạn có thể đã thiết lập proxy web như một phần của cơ sở hạ tầng mạng cục bộ để phục vụ nhiều người dùng để các tài nguyên phổ biến được sử dụng lại một số lần, giảm lưu lượng mạng và độ trễ.*



### 3.4.2.2. HTTP Caching:

#### 1) Khái niệm:

Để hiểu rõ về HTTP caching, đầu tiên ta nên biết đến HTTP, là giao thức truyền lớp ứng dụng dựa trên văn bản và được coi là cơ sở giao tiếp dữ liệu giữa các thiết bị mạng, nghĩa là giữa máy khách (client) và máy chủ (server).

HTTP caching là kỹ thuật lưu trữ tạm thời các phiên bản của các tài nguyên web như HTML, CSS, JavaScript, hình ảnh và các tài liệu khác trên bộ nhớ đệm, có thể là phần cứng hoặc phần mềm. Khi người dùng truy cập một trang web, trình duyệt của họ sẽ lưu trữ các tài nguyên này trong cache để có thể sử dụng lại nhanh chóng trong tương lai. Điều này giúp cải thiện hiệu suất và giảm thời gian tải trang, đặc biệt là khi người dùng truy cập lại cùng một trang web nhiều lần.

Kỹ thuật HTTP caching chính là việc bạn chuyển một bản copy các tài nguyên tĩnh phía server xuống lưu ở dưới client. Về cơ bản, người dùng sẽ cảm nhận thấy một độ trễ rất thấp khi yêu cầu các tài nguyên tĩnh này từ phía Server, lưu lượng truyền đi ít hơn, số yêu cầu đến Server ít hơn, do vậy Server sẽ nhàn hơn để dùng sức của mình vào những việc khác.

#### 2) Lợi ích:

- **Giảm độ trễ:**

Giảm thời gian tải trang ban đầu: Khi truy cập trang web lần đầu, trình duyệt cần tải xuống tất cả các tài nguyên như HTML, CSS, JavaScript, hình ảnh... Điều này có thể dẫn đến thời gian tải trang lâu, đặc biệt đối với người dùng có kết nối internet chậm. Tuy nhiên, với HTTP caching, các tài nguyên này sẽ được lưu trữ trên bộ nhớ đệm của trình duyệt hoặc máy chủ proxy. Khi người dùng truy cập lại trang web, trình duyệt sẽ sử dụng các tài nguyên được lưu trữ trong cache thay vì tải xuống lại từ máy chủ web gốc. Nhờ vậy, thời gian tải trang ban đầu được giảm thiểu đáng kể, mang lại trải nghiệm mượt mà hơn cho người dùng.

Giảm thời gian tải các thành phần trang web: Khi người dùng tương tác với trang web, ví dụ như nhấp chuột vào liên kết hoặc cuộn trang, trình duyệt cần tải xuống các tài nguyên bổ sung. Với HTTP caching, các tài nguyên này cũng có thể được lưu trữ trong cache, giúp giảm thời gian chờ đợi và cải thiện khả năng phản hồi của trang web.

- **Giảm mức tiêu thụ băng thông:**

**Giảm lượng dữ liệu tải xuống:** Khi truy cập trang web, trình duyệt cần tải xuống các tài nguyên tĩnh như hình ảnh, CSS, JavaScript... Những tài nguyên này thường chiếm phần lớn dung lượng trang web. Với HTTP caching, trình duyệt sẽ sử dụng các tài nguyên được lưu trữ trong cache thay vì tải xuống lại từ máy chủ web gốc. Nhờ vậy, lượng dữ liệu cần tải xuống được giảm thiểu đáng kể, tiết kiệm băng thông cho người

dùng. Điều này sẽ giúp tải cho mạng trong trường hợp có nhiều người truy cập cùng lúc. Điều này góp phần giảm tắc nghẽn mạng và đảm bảo trải nghiệm truy cập mượt mà cho tất cả người dùng.

**Giảm số lượng truy vấn đến máy chủ web:** Khi người dùng sử dụng các tài nguyên được lưu trữ trong cache, trình duyệt sẽ không cần phải gửi yêu cầu đến máy chủ web gốc để tải xuống các tài nguyên đó. Điều này giúp giảm số lượng truy vấn đến máy chủ web, giảm tải cho máy chủ và giúp cải thiện hiệu suất tổng thể.

**Giảm tắc nghẽn mạng:** Việc giảm lượng dữ liệu tải xuống cũng giúp giảm tải cho mạng, đặc biệt là trong trường hợp có nhiều người truy cập cùng lúc. Điều này góp phần giảm tắc nghẽn mạng và đảm bảo trải nghiệm truy cập mượt mà cho tất cả người dùng.

**Tiết kiệm chi phí dữ liệu:** Đối với người dùng có gói dữ liệu di động hạn chế, việc giảm mức tiêu thụ băng thông giúp tiết kiệm chi phí sử dụng dữ liệu.

**Cải thiện trải nghiệm người dùng:** Người dùng có kết nối internet chậm thường gặp phải tình trạng tải trang web chậm chạp, ảnh hưởng đến trải nghiệm sử dụng. HTTP caching giúp giảm thời gian chờ đợi, tăng tốc độ tải trang và mang lại trải nghiệm mượt mà hơn cho người dùng.

- ***Giảm số lần truy cập máy chủ:***

Khi người dùng sử dụng các tài nguyên được lưu trữ trong cache, trình duyệt sẽ không cần phải gửi yêu cầu HTTP đến máy chủ web gốc để tải xuống các tài nguyên đó. Điều này giúp giảm tải cho máy chủ web và giảm số lượng truy vấn mạng cần thiết.

**Tóm lại,** HTTP caching mang lại nhiều lợi ích cho cả người dùng và nhà phát triển web. Nhờ lưu trữ các tài nguyên trang web trong bộ nhớ đệm, HTTP caching giúp giảm thời gian tải trang, giảm mức tiêu thụ băng thông, giảm số lần truy cập máy chủ và cải thiện hiệu suất tổng thể của trang web. Việc sử dụng HTTP caching là một giải pháp hiệu quả để nâng cao trải nghiệm người dùng và hiệu quả hoạt động của trang web.

### 3) HTTP Caching hoạt động thế nào?

- **Quyết định lưu trữ**
- Server dùng các HTTP header như Cache-Control, Expires để cho phép hoặc từ chối cache.
- Trình duyệt cũng có thể quyết định dựa trên cài đặt người dùng và loại tài nguyên.
- **Lưu trữ**

- Nếu tài nguyên đã có trong cache → sử dụng luôn, không cần gửi request mới.
- Nếu chưa có → gửi request lên server để tải.

#### • Cập nhật

- **Validation:** Dùng các header như If-Modified-Since để kiểm tra nếu tài nguyên thay đổi.
- **Expiration:** Tài nguyên có thể có thời hạn sử dụng (max-age, Expires) sau đó sẽ được tải lại từ server.

### 3.4.3. Local Storage:

#### 1) Định nghĩa:

Local storage là một phần của trình duyệt web được sử dụng để lưu trữ dữ liệu trực tiếp trên thiết bị của người dùng, như máy tính hoặc điện thoại di động. Dữ liệu được lưu trữ trong local storage có thể được truy cập và cập nhật từ các ứng dụng web mà không cần kết nối mạng, và thông thường được sử dụng để lưu trữ thông tin như cài đặt người dùng, lịch sử truy cập, dữ liệu tạm thời và các dữ liệu khác mà không cần phải truy cập từ máy chủ.

Local storage thường được sử dụng như một phương tiện để cải thiện trải nghiệm người dùng và tăng tốc độ truy cập dữ liệu trong các ứng dụng web.

#### 2) Vai trò và ý nghĩa:

**Tăng tốc độ truy cập dữ liệu:** Bằng cách lưu trữ dữ liệu trên thiết bị của người dùng, local storage giúp giảm thời gian phản hồi của ứng dụng bằng cách cho phép truy cập nhanh chóng đến thông tin cần thiết mà không cần phải gửi yêu cầu tới máy chủ.

**Cải thiện trải nghiệm người dùng:** Với khả năng lưu trữ cài đặt người dùng, lịch sử truy cập, và dữ liệu tạm thời, local storage giúp cá nhân hóa trải nghiệm người dùng, tạo ra một môi trường sử dụng ứng dụng web thuận tiện và linh hoạt hơn.

**Giảm tải cho máy chủ:** Bằng cách giữ một phần dữ liệu ở phía máy khách, local storage giúp giảm tải cho máy chủ bằng cách giảm số lượng yêu cầu được gửi từ trình duyệt web tới máy chủ.

**Hỗ trợ làm việc offline:** Local storage cung cấp khả năng lưu trữ dữ liệu trên thiết bị người dùng, cho phép ứng dụng web hoạt động một cách liên tục mà không cần kết nối mạng, cải thiện trải nghiệm người dùng trong các điều kiện mạng không ổn định hoặc khi không có kết nối internet.

**Bảo mật thông tin nhạy cảm:** Mặc dù local storage không thích hợp cho việc lưu trữ thông tin nhạy cảm như mật khẩu, nhưng nó vẫn có thể được sử dụng để lưu trữ các

thông tin không nhạy cảm mà cần được lưu trữ cục bộ, giúp bảo vệ dữ liệu của người dùng một cách an toàn hơn.

Local storage đóng vai trò quan trọng trong việc tối ưu hóa trải nghiệm người dùng, cải thiện hiệu suất ứng dụng web, giảm tải cho máy chủ và hỗ trợ làm việc offline, đồng thời cũng cần được sử dụng một cách cân nhắc để đảm bảo an toàn và bảo mật cho dữ liệu của người dùng.

### 3) Cơ chế hoạt động:

- **Lưu trữ dữ liệu dạng key-value:**
  - Dữ liệu được lưu dưới dạng chuỗi (string) và được truy cập thông qua các phương thức đơn giản.
  - Dữ liệu chỉ khả dụng trong cùng domain (origin).
    - **Dữ liệu được lưu vĩnh viễn:**
      - Không bị mất khi reload trang, đóng tab, hoặc đóng trình duyệt.
      - Chỉ mất khi người dùng hoặc mã JavaScript xóa đi.
        - **Không đồng bộ giữa tab khác:**
          - Dữ liệu lưu ở tab A có thể được truy cập từ tab B (cùng domain), nhưng không tự đồng bộ thời gian thực (phải reload hoặc dùng storage event).
      - **Không gửi lên server:**
        - Không giống cookie, localStorage không đi kèm request HTTP → bảo mật hơn và giảm tải mạng.
      - **Giới hạn dung lượng:**
        - Mỗi domain thường được cấp khoảng 5MB dữ liệu (khác nhau giữa các trình duyệt).

Local storage cung cấp các phương thức hoạt động để quản lý việc lưu trữ dữ liệu cho các ứng dụng web:

- **localStorage.setItem(key, value)**
  - Chức năng: Lưu một giá trị với tên khóa (key) cụ thể.
  - Giá trị luôn được lưu dưới dạng chuỗi (string).
  - Nếu key đã tồn tại, giá trị sẽ bị ghi đè.
- **localStorage.getItem(key)**
  - Chức năng: Lấy giá trị tương ứng với key đã lưu.
  - Trả về null nếu không tìm thấy key.

- **localStorage.removeItem(key)**
  - Chức năng: Xóa giá trị tương ứng với key chỉ định.
- **localStorage.clear()**
  - Chức năng: Xóa tất cả dữ liệu đã lưu trong localStorage của domain hiện tại.
- **localStorage.key(index)**
  - Chức năng: Truy xuất tên key theo chỉ số (theo thứ tự lưu trữ).
  - Trả về null nếu index vượt quá số lượng key hiện có.
- **localStorage.length**
  - Thuộc tính: Trả về tổng số key hiện có trong localStorage.

### 3.5. Back-end:

#### 3.5.1. In-memory Cache:

##### 1) Định nghĩa:

Khi hệ thống lưu trữ dữ liệu trong RAM, nó được gọi là bộ nhớ đệm trong bộ nhớ (inmemory cache). Đây là dạng bộ nhớ đệm đơn giản nhất so với các hình thức bộ nhớ đệm khác. Nó nằm giữa các ứng dụng và cơ sở dữ liệu để cung cấp phản hồi với tốc độ cao bằng cách lưu trữ dữ liệu từ các yêu cầu trước đó hoặc sao chép trực tiếp từ cơ sở dữ liệu. Với bộ nhớ đệm trong bộ nhớ, thời gian cần thiết để truy cập bộ nhớ cho các yêu cầu I/O và yêu cầu phụ thuộc vào CPU (CPU – bound request) của ứng dụng hoặc máy tính được giảm đáng kể.

Lợi ích lớn nhất của bộ nhớ đệm trong bộ nhớ là nó loại bỏ sự chậm trễ khi một ứng dụng dựa trên cơ sở dữ liệu đĩa phải truy xuất dữ liệu từ đĩa trước khi xử lý.

Bộ nhớ đệm trong bộ nhớ tránh độ trễ và cải thiện hiệu suất ứng dụng trực tuyến vì RAM có tốc độ truy xuất cực nhanh so với đĩa cứng. Theo cách tiếp cận này, dữ liệu được lưu trữ trong bộ nhớ đệm dựa trên cơ sở dữ liệu khóa-giá trị.

Cơ sở dữ liệu được lưu trữ dưới dạng một tập hợp các cặp key – value, và khóa (key) được biểu thị bằng một giá trị duy nhất, trong khi giá trị (value) là dữ liệu được lưu trong bộ nhớ đệm. Mỗi mẫu dữ liệu được xác định bằng một giá trị duy nhất, điều này làm cho quá trình nhanh hơn và hiệu quả hơn.

##### 2) Lợi ích của In-memory cache:

**Tăng tốc độ truy xuất dữ liệu:** Truy cập dữ liệu từ bộ nhớ chính (RAM) nhanh hơn nhiều so với truy cập từ nguồn lưu trữ chính như ổ đĩa cứng hay cơ sở dữ liệu. Bằng cách lưu trữ dữ liệu phổ biến hoặc được truy cập thường xuyên trong bộ nhớ cache, các hoạt động truy cập dữ liệu trở nên nhanh chóng và hiệu quả hơn.

**Giảm tải cho nguồn lưu trữ chính:** Bằng cách giảm số lượng truy cập đến nguồn lưu trữ chính như cơ sở dữ liệu hoặc ổ đĩa, in-memory cache giúp giảm tải cho các nguồn tài nguyên chính và giúp hệ thống xử lý được nhiều yêu cầu hơn cùng một lúc.

**Cải thiện hiệu suất tổng thể:** Thời gian phản hồi nhanh hơn và tốc độ truy xuất dữ liệu tăng cường làm cho hệ thống hoạt động mượt mà hơn, cung cấp trải nghiệm tốt hơn cho người dùng và giảm nguy cơ gây ra sự chậm trễ.

**Hỗ trợ mở rộng:** In-memory cache có thể được mở rộng một cách dễ dàng bằng cách thêm các nút cache mới vào cụm cache hiện có, giúp hệ thống mở rộng linh hoạt và đáp ứng được tải cao.

**Hỗ trợ tính toàn vẹn dữ liệu:** In-memory cache có thể được cấu hình để lưu trữ bản sao dữ liệu hoặc sử dụng các cơ chế như đồng bộ hóa và bảo vệ dữ liệu, giúp đảm bảo tính toàn vẹn của dữ liệu trong hệ thống.

### 3) Cơ chế hoạt động:

- **Truy xuất cache miss:**
  - Kiểm tra xem trong RAM (cache) đã có dữ liệu chưa -> Nếu dữ liệu chưa tồn tại trong cache, kiểm tra trong nguồn chính.
  - Dữ liệu sau đó sẽ được lưu trữ vào RAM cho lần sau.
- **Truy xuất cache hit:**
  - Nếu dữ liệu đã có trong cache thì truy xuất thẳng đến cache mà không cần kiểm tra nguồn.
- **Lưu trữ Key/Value:**
  - Dữ liệu được lưu dưới dạng key / value trong RAM.
  - Hệ thống thường sử dụng **Hash Map** để truy xuất nhanh.
- **Expiration (TTL):**
  - Mỗi mục trong cache thường có thời gian sống giới hạn (TTL).
  - Khi hết hạn thì mục sẽ tự động bị xóa cache để dữ liệu không quá cũ.

### 4) Các loại In-memory cache phổ biến:

#### 4) Local cache:

Local Cache là một phần của bộ nhớ được lưu trữ trực tiếp trên một nút hoặc một ứng dụng cụ thể trong một hệ thống phân tán. Mỗi nút hoặc ứng dụng sẽ duy trì một bản sao của local cache của chính nó. Dữ liệu được lưu trữ trong local cache chỉ có thể được truy cập bởi nút hoặc ứng dụng đó và không được chia sẻ với các nút hoặc ứng dụng khác trong hệ thống.

## **- Cách sử dụng local cache:**

### **• Caching các kết quả truy vấn:**

Trong các ứng dụng web, local cache thường được sử dụng để lưu trữ kết quả của các truy vấn cơ sở dữ liệu hoặc kết quả tính toán phức tạp. Khi một truy vấn được thực hiện, ứng dụng sẽ trước tiên kiểm tra xem kết quả đã được lưu trong local cache chưa. Nếu có, nó sẽ trả về kết quả từ cache mà không cần thực hiện truy vấn lại.

### **• Lưu trữ dữ liệu tạm thời:**

Local cache cũng có thể được sử dụng để lưu trữ dữ liệu tạm thời như phiên đăng nhập, thông tin người dùng hoặc các cấu trúc dữ liệu nhất định để giảm thiểu thời gian truy xuất và tải lại từ nguồn lưu trữ chính.

### **• Cache các tài nguyên tĩnh:**

Trong các ứng dụng web, local cache thường được sử dụng để lưu trữ các tài nguyên tĩnh như hình ảnh, CSS và JavaScript files. Việc lưu trữ những tài nguyên này trong cache giúp giảm thiểu thời gian tải trang và tăng trải nghiệm người dùng.

### **• Cache các kết quả xử lý:**

Trong các ứng dụng có yêu cầu tính toán phức tạp, local cache có thể được sử dụng để lưu trữ các kết quả xử lý trước đó. Việc này giúp tránh việc tính toán lại những kết quả đã được tính toán trước đó và giảm thiểu thời gian xử lý.

## **- Lợi ích của Local Cache:**

### **• Tăng hiệu suất ứng dụng:**

Sử dụng local cache giúp tăng hiệu suất của ứng dụng bằng cách giảm thiểu thời gian truy cập dữ liệu từ nguồn lưu trữ chính. Việc sử dụng cache cho phép ứng dụng xử lý các yêu cầu của người dùng một cách nhanh chóng và hiệu quả hơn, cải thiện trải nghiệm người dùng cuối.

### **• Giảm tải cho hệ thống:**

Sử dụng local cache giúp giảm tải cho hệ thống bằng cách giảm số lượng truy vấn đến nguồn lưu trữ chính. Thay vì phải truy cập trực tiếp vào cơ sở dữ liệu hoặc hệ thống tệp mỗi khi có yêu cầu, dữ liệu có thể được cung cấp từ bộ nhớ địa phương một cách nhanh chóng.

### **• Tăng tính linh động:**

Local cache cung cấp một cơ chế linh động để quản lý và lưu trữ dữ liệu. Bằng cách lưu trữ dữ liệu tạm thời trong bộ nhớ địa phương, hệ thống có thể dễ dàng cập nhật và truy cập dữ liệu mà không cần phải truy cập lại các nguồn lưu trữ chính. Điều này giúp cải thiện khả năng mở rộng và bảo trì của hệ thống.

- **Giảm chi phí tài nguyên:**

Sử dụng local cache giúp giảm sự phụ thuộc vào các nguồn tài nguyên như cơ sở dữ liệu hoặc hệ thống tệp. Bằng cách lưu trữ dữ liệu trực tiếp trong bộ nhớ địa phương, local cache giảm chi phí và tài nguyên mà hệ thống cần để truy cập và quản lý dữ liệu từ các nguồn lưu trữ chính.

## 2) **Distributed Cache:**

Distributed cache là một hệ thống cache phân tán, tức là dữ liệu được lưu trữ và quản lý trên nhiều nút (node) khác nhau trong mạng. Mỗi nút trong hệ thống distributed cache chịu trách nhiệm lưu trữ một phần của dữ liệu và chia sẻ nó với các nút khác trong cụm.

*Cách sử dụng distributed cache và lợi ích của nó như sau:*

- **Tăng hiệu suất ứng dụng và giảm thời gian truy cập dữ liệu:**

Sử dụng distributed cache giúp tăng hiệu suất của ứng dụng bằng cách giảm thiểu thời gian truy cập dữ liệu từ nguồn lưu trữ chính. Việc truy cập dữ liệu từ bộ nhớ địa phương trên các nút trong mạng nhanh hơn so với việc truy cập trực tiếp vào cơ sở dữ liệu hoặc hệ thống tệp. Điều này cải thiện trải nghiệm người dùng và giảm tải cho hệ thống.

- **Cân bằng tải và mở rộng ngang:**

Distributed cache cung cấp cơ chế tự động cân bằng tải giữa các nút trong mạng. Khi có sự thay đổi trong tải công việc hoặc số lượng yêu cầu, hệ thống tự động điều chỉnh việc phân phối dữ liệu và tải công việc để đảm bảo rằng mọi nút trong hệ thống hoạt động hiệu quả.

- **Tính nhất quán và đồng bộ hóa dữ liệu:**

Distributed cache cung cấp các cơ chế đồng bộ hóa dữ liệu giữa các nút trong mạng để đảm bảo tính nhất quán của dữ liệu. Khi dữ liệu được cập nhật trên một nút, các thay đổi sẽ được truyền đến các nút khác trong hệ thống để đảm bảo rằng mọi người dùng đều nhận được phiên bản mới nhất của dữ liệu.

- **Tăng cường khả năng chịu lỗi:**

Distributed cache có khả năng chịu lỗi cao bằng cách sao lưu dữ liệu trên nhiều nút trong mạng. Khi một nút gặp sự cố, các nút khác có thể tiếp tục cung cấp dữ liệu cho các ứng dụng mà không bị gián đoạn.

- **Giảm chi phí tài nguyên:**

Distributed cache giúp giảm sự phụ thuộc vào các nguồn tài nguyên như cơ sở dữ liệu hoặc hệ thống tệp. Việc lưu trữ dữ liệu tạm thời trong bộ nhớ địa phương giảm chi phí



và tài nguyên mà hệ thống cần để truy cập và quản lý dữ liệu từ các nguồn lưu trữ chính.

### 3.5.2. Các best practices cho Caching

#### 1. Lazy caching:

##### 1) Khái niệm:

Chiến lược lazy caching, còn được gọi là lazy population hoặc cache-aside, là một hình thức phổ biến của caching. Đây là chiến lược caching chỉ tải dữ liệu vào cache khi có yêu cầu truy cập. Chiến lược này có ưu điểm là tiết kiệm bộ nhớ và thời gian xử lý, vì chỉ những dữ liệu thực sự cần thiết mới được lưu trữ.

##### 2) Luồng hoạt động:

- Ứng dụng của bạn nhận được truy vấn dữ liệu, ví dụ như 10 tin tức mới nhất.
- Ứng dụng của bạn sẽ kiểm tra bộ nhớ đệm để xem đối tượng có nằm trong bộ nhớ đệm hay không.
- Nếu có (lượt truy cập bộ nhớ đệm), đối tượng được lưu trong bộ nhớ đệm sẽ được trả về và luồng cuộc gọi kết thúc.
- Nếu không (lỗi bộ nhớ đệm), cơ sở dữ liệu sẽ được truy vấn để tìm đối tượng. Bộ nhớ đệm sẽ được điền và đối tượng sẽ được trả về.

#### - Mô tả chi tiết các bước thực hiện:

##### 1) Kiểm tra cache:

Hệ thống cần lựa chọn một cơ chế lưu trữ cache phù hợp với nhu cầu và đặc điểm của ứng dụng. Một số cơ chế lưu trữ cache phổ biến bao gồm:

- **Hash table:** Cung cấp truy cập dữ liệu nhanh chóng và hiệu quả.
- **Memcached:** Một hệ thống lưu trữ bộ nhớ cache phân tán, cung cấp khả năng mở rộng cao.
- **Redis:** Một kho lưu trữ dữ liệu cấu trúc dạng khóa-giá trị, cung cấp nhiều tính năng nâng cao như hỗ trợ dữ liệu cấu trúc và ngôn ngữ kịch bản.

Khóa truy vấn là một chuỗi duy nhất được sử dụng để xác định một đối tượng trong cache. Khóa truy vấn cần được thiết kế cẩn thận để đảm bảo tính nhất quán và hiệu quả. Ví dụ, khóa truy vấn có thể bao gồm ID sản phẩm, URL trang web hoặc một chuỗi duy nhất đại diện cho truy vấn dữ liệu

Khi nhận được yêu cầu truy cập dữ liệu, hệ thống sẽ sử dụng khóa truy vấn để kiểm tra cache. Hệ thống sẽ tra cứu khóa truy vấn trong cơ chế lưu trữ cache để xác định xem đối tượng có được lưu trữ hay không

## 2) Xử lý cache hit:

Trước khi trả về đối tượng từ cache, hệ thống cần xác minh xem đối tượng có còn hợp lệ hay không. Việc xác minh tính hợp lệ có thể dựa trên các yếu tố như thời gian lưu trữ cache, phiên bản dữ liệu hoặc các thông tin khác liên quan đến trạng thái của đối tượng.

Nếu đối tượng hợp lệ, hệ thống sẽ trả về đối tượng cho ứng dụng. Việc trả về đối tượng có thể được thực hiện trực tiếp hoặc thông qua một giao diện lập trình ứng dụng (API) cụ thể.

## 3) Xử lý cache miss:

Nếu đối tượng không được tìm thấy trong cache, hệ thống sẽ truy vấn cơ sở dữ liệu để lấy dữ liệu cần thiết. Việc truy vấn cơ sở dữ liệu có thể được thực hiện bằng cách sử dụng SQL hoặc các ngôn ngữ truy vấn dữ liệu khác.

Hệ thống sẽ lấy kết quả trả về từ truy vấn cơ sở dữ liệu và lưu trữ dữ liệu trong một biến tạm thời.

Hệ thống sẽ lưu trữ dữ liệu từ biến tạm thời vào cache, sử dụng khóa truy vấn tương ứng. Việc lưu trữ dữ liệu vào cache có thể bao gồm việc thiết lập thời gian lưu trữ, phiên bản dữ liệu và các thông tin khác liên quan đến trạng thái của đối tượng.

Trả về đối tượng: Hệ thống sẽ trả về dữ liệu từ biến tạm thời cho ứng dụng.

## 3) Ưu và nhược điểm khi sử dụng Lazy Caching:

### - Ưu điểm:

Lazy caching là một chiến lược lưu trữ dữ liệu thông minh, mang lại nhiều lợi ích thiết thực cho hiệu suất và khả năng mở rộng của ứng dụng. Dựa trên những ưu điểm đã trình bày, hãy cùng khám phá chi tiết hơn về sức mạnh của lazy caching trong các khía cạnh sau:

### 1) *Tối ưu hóa hiệu suất ứng dụng:*

**Giảm tải cho cơ sở dữ liệu:** Lazy caching hoạt động như một trung gian đắc lực, giúp giảm thiểu số lượng truy vấn trực tiếp đến cơ sở dữ liệu. Khi dữ liệu cần thiết được tìm thấy trong cache, ứng dụng có thể truy cập ngay lập tức mà không cần tốn thời gian chờ đợi truy vấn được thực hiện. Điều này giúp giảm tải đáng kể cho cơ sở dữ liệu, đặc biệt là trong các hệ thống có lưu lượng truy cập cao.

**Cải thiện thời gian phản hồi:** Nhờ việc giảm tải cho cơ sở dữ liệu và truy cập dữ liệu nhanh chóng từ cache, lazy caching giúp ứng dụng phản hồi yêu cầu của người dùng một cách nhanh chóng và mượt mà. Người dùng có thể trải nghiệm ứng dụng một cách trơn tru, hạn chế tối đa tình trạng chờ đợi và thất vọng.

**Tiết kiệm băng thông:** Khi dữ liệu được lưu trữ trong cache, ứng dụng không cần tải xuống dữ liệu tương tự nhiều lần từ máy chủ. Điều này giúp tiết kiệm băng thông, đặc biệt hữu ích cho các ứng dụng di động sử dụng mạng di động hoặc các khu vực có kết nối internet hạn chế.

## 2) *Nâng cao khả năng mở rộng:*

**Giảm thiểu điểm nghẽn hệ thống:** Lazy caching giúp phân tán tải trọng truy cập dữ liệu, giảm thiểu nguy cơ xảy ra điểm nghẽn hệ thống, đặc biệt là khi có nhiều người dùng truy cập đồng thời.

**Cải thiện khả năng xử lý truy vấn:** Khi dữ liệu thường xuyên truy cập được lưu trữ trong cache, cơ sở dữ liệu có thể tập trung xử lý các truy vấn phức tạp và quan trọng hơn. Điều này giúp hệ thống hoạt động hiệu quả và ổn định hơn, đặc biệt trong các trường hợp có lượng truy cập dữ liệu lớn và đa dạng.

**Dễ dàng mở rộng lưu trữ:** Lazy caching cho phép sử dụng nhiều máy chủ cache để phân tán dữ liệu, giúp dễ dàng mở rộng dung lượng lưu trữ và đáp ứng nhu cầu truy cập ngày càng tăng của hệ thống.

## 3) *Tăng cường tính linh hoạt và kiểm soát:*

**Quản lý dữ liệu hiệu quả:** Lazy caching cung cấp khả năng kiểm soát chi tiết việc lưu trữ và cập nhật dữ liệu trong cache. Các nhà phát triển có thể linh hoạt xác định dữ liệu nào cần được lưu trữ, thời gian lưu trữ và điều kiện cập nhật cache.

**Tùy chỉnh chiến lược:** Lazy caching cho phép tùy chỉnh chiến lược lưu trữ cache phù hợp với nhu cầu cụ thể của từng ứng dụng. Ví dụ, có thể ưu tiên lưu trữ dữ liệu truy cập thường xuyên hoặc dữ liệu quan trọng đối với hiệu suất ứng dụng.

**Hỗ trợ đa dạng dữ liệu:** Lazy caching có thể lưu trữ nhiều loại dữ liệu khác nhau, bao gồm dữ liệu tĩnh, dữ liệu động, dữ liệu được tính toán hoặc kết quả của các truy vấn phức tạp.

## 4) *Dễ dàng triển khai và quản lý:*

**Tích hợp đơn giản:** Lazy caching có thể được tích hợp dễ dàng vào hầu hết các kiến trúc ứng dụng hiện có mà không yêu cầu thay đổi lớn về mã nguồn hoặc cấu trúc hệ thống.

**Công cụ hỗ trợ đa dạng:** Có nhiều công cụ và thư viện hỗ trợ lazy caching, giúp đơn giản hóa việc triển khai và quản lý cache trong ứng dụng.

**Giám sát hiệu quả:** Lazy caching cung cấp khả năng giám sát hiệu quả hoạt động của cache, giúp xác định các vấn đề tiềm ẩn và tối ưu hóa hiệu suất.

## 5) *Tiết kiệm chi phí:*

**Giảm chi phí hạ tầng:** Việc giảm tải cho cơ sở dữ liệu và tối ưu hóa hiệu suất ứng dụng có thể giúp giảm nhu cầu về tài nguyên phần cứng, dẫn đến tiết kiệm chi phí hạ tầng.

**Cải thiện hiệu suất ứng dụng:** Hiệu suất ứng dụng tốt hơn có thể dẫn đến tăng doanh thu và giảm chi phí vận hành, bảo trì.

**- Nhược điểm:**

Lazy caching, mặc dù mang lại nhiều lợi ích về hiệu suất và tải cho cơ sở dữ liệu, nhưng cũng tiềm ẩn một số bất lợi cần được quản lý cẩn thận để đảm bảo tính nhất quán và hiệu quả của hệ thống.

Một trong những bất lợi chính của lazy caching là khi xảy ra cache miss, tức là dữ liệu yêu cầu không tồn tại trong cache. Khi này, ứng dụng phải thực hiện truy vấn cơ sở dữ liệu để lấy dữ liệu cần thiết, và quá trình này có thể tốn thời gian đáng kể. Thời gian xử lý cache miss kéo dài có thể ảnh hưởng đáng kể đến trải nghiệm người dùng, đặc biệt là trong các ứng dụng yêu cầu độ phản hồi nhanh. Điều này có thể gây khó chịu cho người dùng và làm giảm sự hài lòng với ứng dụng.

Ngoài ra, việc sử dụng lazy caching có thể gây ra hiện tượng cache stampede. Đây là tình huống khi dữ liệu trong cache bị hết hạn hoặc bị xóa và có nhiều yêu cầu đồng thời đến truy vấn dữ liệu mới từ cơ sở dữ liệu. Khi điều này xảy ra, một lượng lớn yêu cầu cùng truy vấn dữ liệu từ cơ sở dữ liệu, gây ra tải lớn đồng thời lên cơ sở dữ liệu và làm tăng thời gian xử lý. Hiện tượng cache stampede có thể dẫn đến tình trạng quá tải và giảm hiệu suất của hệ thống, làm cho ứng dụng trở nên không ổn định và không đáng tin cậy.

Tóm lại, mặc dù lazy caching mang lại nhiều lợi ích về hiệu suất và tải cho hệ thống, nhưng cũng tiềm ẩn một số bất lợi cần được quản lý cẩn thận. Các vấn đề như cache miss, cache stampede, dữ liệu không hợp lệ và độ trễ có thể ảnh hưởng đến hiệu suất, tính nhất quán và trải nghiệm người dùng của ứng dụng. Để vượt qua những bất lợi này, cần có chiến lược phù hợp trong việc quản lý cache, kiểm soát đồng bộ dữ liệu và sử dụng các kỹ thuật tối ưu để giảm thiểu độ trễ.

**Các bước cần thiết khi ứng dụng chạy Lazy Caching vào ứng dụng của bạn:**

**3.1.4.1. Xác định dữ liệu phù hợp:**

**Phân tích truy cập dữ liệu:** Xác định các loại dữ liệu thường xuyên được truy cập bởi ứng dụng. Ưu tiên dữ liệu tĩnh, ít thay đổi và có thời gian truy cập lâu cho Lazy Caching.

**Đánh giá tính chất dữ liệu:** Xem xét kích thước dữ liệu, thời gian cập nhật và mức độ phụ thuộc vào dữ liệu thời gian thực. Tránh lưu trữ dữ liệu quá lớn, cập nhật thường xuyên hoặc phụ thuộc vào tính nhất quán cao vào cache.

**Lựa chọn dữ liệu phù hợp:** Dựa trên phân tích và đánh giá, lựa chọn các loại dữ liệu phù hợp để lưu trữ trong cache, đảm bảo cân bằng giữa lợi ích hiệu suất và độ phức tạp quản lý.

Ví dụ:

**Ứng dụng mạng xã hội:** Thông tin người dùng, bài đăng, hình ảnh ít thay đổi có thể được lưu trữ trong cache.

**Ứng dụng thương mại điện tử:** Thông tin sản phẩm, danh mục sản phẩm, giá cả có thể được lưu trữ trong cache.

**Ứng dụng tin tức:** Bài viết tin tức, hình ảnh, video đã xuất bản có thể được lưu trữ trong cache.

#### 3.1.4.2. Áp dụng chiến lược Lazy Caching:

Sau khi xác định được dữ liệu phù hợp cho lazy caching, bạn cần áp dụng chiến lược để lưu trữ và truy xuất dữ liệu từ cache. Một cách thông thường là sử dụng một hệ thống key-value store để lưu trữ dữ liệu trong cache. Khi có yêu cầu truy xuất dữ liệu, bạn kiểm tra xem dữ liệu có tồn tại trong cache hay không. Nếu có, bạn trả về dữ liệu từ cache. Trong trường hợp cache miss, tức là dữ liệu không tồn tại trong cache, bạn thực hiện truy vấn cơ sở dữ liệu để lấy dữ liệu và sau đó lưu trữ dữ liệu đó vào cache để sử dụng cho các yêu cầu tiếp theo.

#### 3.1.4.3. Lựa chọn công cụ và thư viện hỗ trợ:

Để thực hiện lazy caching trong ứng dụng của bạn, bạn cần lựa chọn các công cụ và thư viện hỗ trợ. Có nhiều lựa chọn phổ biến như Redis, Memcached, hoặc các thư viện caching như Spring Cache (cho Java) hoặc Django Cache (cho Python). Các công cụ và thư viện này cung cấp các chức năng cơ bản để lưu trữ và truy xuất dữ liệu từ cache một cách dễ dàng.

#### 3.1.4.4. Đảm bảo xóa cache và cập nhật đúng:

Một khía cạnh quan trọng của lazy caching là đảm bảo tính nhất quán giữa dữ liệu trong cache và dữ liệu trong cơ sở dữ liệu chính. Khi dữ liệu được cập nhật hoặc xóa khỏi cơ sở dữ liệu, bạn cần đảm bảo rằng cache cũng được cập nhật hoặc xóa tương ứng. Điều này đảm bảo rằng dữ liệu trong cache luôn được cập nhật và chính xác. Một

cách thông thường là sử dụng các sự kiện hoặc ghi nhật ký để phát hiện và xử lý các thay đổi dữ liệu, sau đó cập nhật hoặc xóa dữ liệu tương ứng trong cache.

Trong quá trình triển khai lazy caching, quan trọng là theo dõi hiệu suất của hệ thống và đảm bảo rằng bất lợi của lazy caching không vượt qua lợi ích của nó. Điều này có thể đòi hỏi điều chỉnh và tinh chỉnh các tham số cache như kích thước cache, thời gian hết hạn, và cơ chế xóa cache.

Tóm lại, để sử dụng lazy caching trong ứng dụng của bạn, bạn cần xác định dữ liệu phù hợp cho caching, áp dụng chiến lược caching, lựa chọn công cụ và thư viện hỗ trợ, và đảm bảo việc xóa cache và cập nhật đúng.

### **Một số Open Sources hỗ trợ lazy caching:**

#### **Django (Python web framework):**

- Lazy caching được sử dụng khi truy vấn dữ liệu từ cơ sở dữ liệu và cache thủ công bằng `cache.get()/cache.set()`.
- GitHub: <https://github.com/django/django>

#### **Flask-Caching (Python):**

- Thư viện mở rộng cho Flask để hỗ trợ caching.
- Hỗ trợ lazy caching thông qua decorators như `@cache.memoize()` hoặc kiểm tra thủ công.

#### **Spring Cache (Java / Spring Framework):**

- Cung cấp annotation-based caching, hỗ trợ lazy behavior.
- Khi dùng `@Cacheable`, hệ thống chỉ cache sau lần đầu gọi hàm
- GitHub: <https://github.com/spring-projects/spring-framework>

## **3.6. Write – Through Cache:**

### **3.6.1. Write-Through Cache là gì?**

Write-Through Cache là một chiến lược quản lý cache trong đó dữ liệu được ghi đồng thời vào cả cache và nguồn dữ liệu chính (thường là cơ sở dữ liệu) khi có thay đổi. Điều này đảm bảo rằng dữ liệu trong cache luôn cập nhật nhất quán với dữ liệu trong nguồn chính.

### **3.6.2. Khi nào nên sử dụng Write-Through Cache?**

Chiến lược Write-Through Cache phù hợp trong những trường hợp sau:

***Dữ liệu cần được cập nhật thường xuyên và truy cập thường xuyên:***

Ví dụ: Dữ liệu giá cả sản phẩm trong hệ thống thương mại điện tử, dữ liệu bảng xếp hạng trò chơi, dữ liệu tin tức mới nhất, v.v.

Việc cập nhật dữ liệu thường xuyên đảm bảo rằng dữ liệu trong cache luôn nhất quán với dữ liệu trong nguồn chính, giúp cải thiện hiệu suất truy cập dữ liệu và mang lại trải nghiệm tốt hơn cho người dùng.

***Khi việc đảm bảo tính nhất quán dữ liệu là rất quan trọng:***

Ví dụ: Hệ thống thanh toán ngân hàng, hệ thống quản lý tài chính, hệ thống y tế, v.v.

Trong những trường hợp này, việc đảm bảo dữ liệu trong cache luôn nhất quán với dữ liệu trong nguồn chính là rất quan trọng để tránh sai sót và đảm bảo tính chính xác của thông tin.

***Khi hiệu suất truy cập dữ liệu là yếu tố quan trọng:***

Ví dụ: Ứng dụng web có lưu lượng truy cập cao, hệ thống xử lý giao dịch trực tuyến, v.v.

Write-Through Cache giúp giảm thiểu khả năng xảy ra đọc trượt cache, từ đó cải thiện hiệu suất truy cập dữ liệu và đáp ứng nhu cầu truy cập nhanh chóng của người dùng.

**Một số ứng dụng cụ thể của Write-Through Cache:**

**Hệ thống thương mại điện tử:** Cập nhật giá cả sản phẩm, thông tin sản phẩm mới, giỏ hàng mua sắm, v.v.

**Hệ thống mạng xã hội:** Cập nhật bài đăng mới, thông tin người dùng, lượt thích, bình luận, v.v.

**Hệ thống tin tức:** Cập nhật tin tức mới nhất, bài báo mới, v.v.

**Hệ thống trò chơi:** Cập nhật bảng xếp hạng trò chơi, điểm số người chơi, trạng thái trò chơi, v.v.

**Hệ thống thanh toán:** Cập nhật số dư tài khoản, lịch sử giao dịch, thông tin thanh toán, v.v.

**Hệ thống quản lý đặt chỗ và đặt vé:** Cập nhật thông tin đặt chỗ cho chuyến bay, vé xem phim, vé sự kiện, v.v.

**Hệ thống quản lý thư viện số:** Cập nhật thông tin sách mới, tác giả, danh mục sách, v.v.

**Hệ thống quản lý đồng bộ hóa dữ liệu:** Cập nhật thông tin về các thay đổi dữ liệu như cập nhật danh sách liên lạc, lịch làm việc, tệp tin, v.v.

**Hệ thống quản lý trực tuyến:** Cập nhật thông tin đặt chỗ, giá cả và trạng thái phòng hoặc dịch vụ.

**Hệ thống quản lý tài nguyên:** Cập nhật thông tin về tình trạng sử dụng, lịch sử bảo trì và trạng thái các tài nguyên.

**Hệ thống quản lý địa điểm:** Cập nhật thông tin về địa điểm mới, đánh giá, đề xuất địa điểm, v.v.

**Hệ thống quản lý khoá học trực tuyến:** Cập nhật thông tin về khóa học mới, tiến độ học tập, điểm số, v.v.

**Hệ thống quản lý sự kiện:** Cập nhật thông tin về sự kiện mới, lịch trình, đăng ký, v.v.

Các ứng dụng này đều có yêu cầu cao về tính nhất quán và đồng bộ giữa cache và cơ sở dữ liệu. Sử dụng Write-Through Cache trong các trường hợp này giúp đảm bảo rằng dữ liệu luôn được cập nhật đồng bộ và tăng hiệu suất truy xuất dữ liệu.

### **3.6.3. Cơ chế hoạt động của Write-Through Cache:**

Write-Through Cache là một thành phần quan trọng trong kiến trúc bộ nhớ đệm để tăng cường hiệu suất truy xuất dữ liệu. Nó được sử dụng rộng rãi trong các hệ thống máy tính và các ứng dụng mà đòi hỏi tính nhất quán giữa cache và cơ sở dữ liệu chính. Mục tiêu của Write-

Through Cache là đảm bảo rằng mọi thay đổi dữ liệu được ghi vào cache đều được ghi vào cơ sở dữ liệu chính ngay lập tức.

Khi một yêu cầu ghi dữ liệu được gửi đến Write-Through Cache, quá trình ghi dữ liệu diễn ra theo các bước sau:

**Nhận yêu cầu ghi dữ liệu:** Khi cache nhận được yêu cầu ghi dữ liệu, nó kiểm tra xem dữ liệu có tồn tại trong cache hay không. Nếu dữ liệu đã có trong cache, tiến trình ghi dữ liệu sẽ được tiếp tục. Ngược lại, nếu dữ liệu không có trong cache, cache phải tìm dữ liệu trong cơ sở dữ liệu chính.

**Tìm kiếm dữ liệu trong cache:** Nếu dữ liệu đã tồn tại trong cache, cache sẽ xác định vị trí của dữ liệu đó để tiến hành cập nhật. Công việc này thường được thực hiện bằng cách sử dụng các kỹ thuật như bộ nhớ cache tương tự (associative cache) hoặc bảng băm (hash table) để tìm kiếm nhanh chóng.

**Cập nhật dữ liệu trong cache:** Sau khi xác định được vị trí của dữ liệu trong cache, cache sẽ cập nhật dữ liệu này với giá trị mới từ yêu cầu ghi dữ liệu. Quá trình cập nhật này thường bao gồm việc ghi đè (overwrite) dữ liệu cũ trong cache bằng dữ liệu mới. Quá trình này đảm bảo rằng cache luôn giữ các giá trị dữ liệu mới nhất.

**Gửi yêu cầu ghi dữ liệu đến cơ sở dữ liệu chính:** Sau khi cập nhật dữ liệu trong cache, cache sẽ gửi yêu cầu ghi dữ liệu tương ứng đến cơ sở dữ liệu chính. Yêu cầu này chứa thông tin về vị trí của dữ liệu trong cơ sở dữ liệu và giá trị mới của dữ liệu.



**Cập nhật dữ liệu trong cơ sở dữ liệu chính:** Cơ sở dữ liệu chính nhận yêu cầu ghi dữ liệu từ cache và tiến hành cập nhật dữ liệu tương ứng. Quá trình này đảm bảo rằng dữ liệu trong cơ sở dữ liệu chính được cập nhật với giá trị mới từ cache. Điều này đảm bảo tính nhất quán giữa cache và cơ sở dữ liệu chính.

**Xác nhận hoàn tất ghi dữ liệu:** Sau khi cơ sở dữ liệu chính hoàn tất việc cập nhật dữ liệu, nó sẽ gửi một xác nhận về thành công hoặc thất bại cho cache. Nếu việc cập nhật thành công, cache sẽ tiếp tục xử lý các yêu cầu tiếp theo. Nếu việc cập nhật thất bại, cache có thể thực hiện các biện pháp khắc phục như gửi lại yêu cầu ghi dữ liệu hoặc thông báo lỗi cho hệ thống.

### **3.6.4. Ưu và nhược điểm của Write-Through Cache:**

#### **1) Ưu điểm:**

Tránh đọc trượt cache, đảm bảo truy cập dữ liệu nhanh chóng và nhất quán: Khi dữ liệu được ghi vào cache thông qua Write-Through Cache, cache được cập nhật ngay lập tức. Điều này đảm bảo rằng các truy cập dữ liệu tiếp theo sẽ nhận được dữ liệu mới nhất từ cache, tránh trường hợp đọc trượt (cache miss). Việc tránh đọc trượt giúp cải thiện hiệu suất truy xuất dữ liệu và đảm bảo tính nhất quán giữa cache và cơ sở dữ liệu chính.

Cải thiện hiệu suất ứng dụng, đặc biệt cho các trường hợp dữ liệu được truy cập thường xuyên: Khi dữ liệu được ghi vào cache, các truy cập dữ liệu sau đó có thể được thực hiện nhanh chóng từ cache mà không cần truy cập cơ sở dữ liệu chính. Điều này giúp cải thiện hiệu suất của ứng dụng, đặc biệt là trong các trường hợp mà dữ liệu được truy cập thường xuyên.

Đơn giản hóa việc quản lý cache do dữ liệu luôn được cập nhật theo thời gian thực: Với Write-Through Cache, dữ liệu trong cache luôn được cập nhật theo thời gian thực. Khi có yêu cầu ghi dữ liệu, cache sẽ cập nhật dữ liệu đó ngay lập tức và gửi yêu cầu cập nhật tương ứng đến cơ sở dữ liệu chính. Điều này đơn giản hóa việc quản lý cache vì không cần quan tâm đến việc đồng bộ hóa cache và cơ sở dữ liệu chính.

#### **2) Nhược điểm:**

Tăng nguy cơ mất dữ liệu nếu xảy ra lỗi cache trước khi dữ liệu được ghi vào cơ sở dữ liệu: Trong Write-Through Cache, việc ghi dữ liệu trực tiếp vào cache có nguy cơ mất dữ liệu nếu xảy ra lỗi cache trước khi dữ liệu được ghi vào cơ sở dữ liệu chính. Nếu cache gặp sự cố hoặc mất điện, dữ liệu ghi vào cache nhưng chưa được ghi vào cơ sở dữ liệu chính có thể bị mất. Điều này đòi hỏi các biện pháp bảo mật và khôi phục dữ liệu phù hợp để xử lý các tình huống này.

Gây ra tình trạng ghi nhiều vào cache, đặc biệt cho các trường hợp dữ liệu được cập nhật liên tục: Với Write-Through Cache, mọi yêu cầu ghi dữ liệu đều phải cập nhật cả cache và cơ sở dữ liệu chính. Điều này có thể dẫn đến tình trạng ghi nhiều vào cache,

đặc biệt là trong các trường hợp mà dữ liệu được cập nhật liên tục. Việc ghi nhiều vào cache có thể làm tăng tải cho hệ thống và ảnh hưởng đến hiệu suất chung.

Tiêu tốn thêm tài nguyên hệ thống do cần cập nhật đồng thời cả cache và cơ sở dữ liệu: Với Write-Through Cache, mỗi lần ghi dữ liệu đều yêu cầu cập nhật cả cache và cơ sở dữ liệu chính. Điều này tiêu tốn thêm tài nguyên hệ thống, bao gồm băng thông mạng và tài nguyên xử lý. Việc cập nhật đồng thời cả cache và cơ sở dữ liệu có thể tạo ra tải cho hệ thống và ảnh hưởng đến hiệu suất chung của ứng dụng.

Tóm lại, việc sử dụng Write-Through Cache mang lại các lợi ích như tránh đọc trượt cache, cải thiện hiệu suất ứng dụng và đơn giản hóa việc quản lý cache. Tuy nhiên, cần lưu ý các hạn chế tiềm ẩn như nguy cơ mất dữ liệu, tình trạng ghi nhiều vào cache và tiêu tốn thêm tài nguyên hệ thống. Việc áp dụng Write-Through Cache cần được xem xét kỹ lưỡng để đảm bảo rằng các ưu điểm vượt qua các nhược điểm và phù hợp với yêu cầu và môi trường cụ thể của hệ thống.

### **3.6.5. Các phương pháp để triển khai chiến lược Write – Through Cache**

hiệu quả:

Write-Through Cache là chiến lược quản lý cache trong đó dữ liệu được ghi đồng thời vào cả cache và nguồn dữ liệu chính (thường là cơ sở dữ liệu) khi có thay đổi. Việc này đảm bảo rằng dữ liệu trong cache luôn cập nhật nhất quán với dữ liệu trong nguồn chính, giúp cải thiện hiệu suất truy cập dữ liệu và mang lại trải nghiệm tốt hơn cho người dùng. Tuy nhiên, để triển khai và sử dụng Write-Through Cache hiệu quả, cần áp dụng các phương pháp hay nhất sau:

#### ***Lựa chọn kích thước cache phù hợp:***

**Bối cảnh:** Kích thước cache ảnh hưởng trực tiếp đến hiệu suất truy cập dữ liệu và dung lượng bộ nhớ hệ thống. Cache quá nhỏ có thể dẫn đến nhiều lần truy cập cơ sở dữ liệu, giảm hiệu suất.

Cache quá lớn có thể chiếm dụng nhiều bộ nhớ, ảnh hưởng đến hiệu suất hệ thống khác.

#### **Phương pháp:**

- Xác định nhu cầu truy cập dữ liệu: Phân tích tần suất truy cập, loại dữ liệu truy cập và dung lượng dữ liệu cần lưu trữ.
- Đánh giá dung lượng bộ nhớ hệ thống: Xác định lượng bộ nhớ sẵn có để phân bổ cho cache.
- Sử dụng các công cụ đo lường: Theo dõi hiệu suất cache và điều chỉnh kích thước cho phù hợp.

Ví dụ:

Ứng dụng web thương mại điện tử: Nên sử dụng cache có kích thước lớn để lưu trữ dữ liệu sản phẩm, giá cả, giỏ hàng mua sắm, v.v. vì những dữ liệu này được truy cập thường xuyên.

Ứng dụng xử lý giao dịch: Nên sử dụng cache có kích thước nhỏ hơn để lưu trữ dữ liệu lịch sử giao dịch vì những dữ liệu này không được truy cập thường xuyên.

### ***Áp dụng chiến lược chống lỗi cache:***

**Bối cảnh:** Lỗi cache có thể dẫn đến truy cập dữ liệu sai lệch, ảnh hưởng đến tính toàn vẹn dữ liệu.

#### **Phương pháp:**

- Cập nhật đồng bộ: Việc cập nhật dữ liệu trong cache và cơ sở dữ liệu diễn ra đồng thời để đảm bảo tính nhất quán.
- Cập nhật theo thời gian: Cập nhật dữ liệu trong cache định kỳ hoặc khi có thay đổi quan trọng trong cơ sở dữ liệu. o Cơ chế xác minh: Kiểm tra tính nhất quán dữ liệu giữa cache và cơ sở dữ liệu trước khi sử dụng.

Ví dụ:

**Cập nhật đồng bộ:** Sử dụng cơ chế khóa để đảm bảo rằng chỉ có một phiên bản dữ liệu được cập nhật trong cache và cơ sở dữ liệu tại một thời điểm.

**Cập nhật theo thời gian:** Cập nhật dữ liệu trong cache mỗi phút để đảm bảo dữ liệu luôn cập nhật trong vòng một phút.

**Cơ chế xác minh:** Sử dụng checksum để kiểm tra tính nhất quán dữ liệu giữa cache và cơ sở dữ liệu trước khi sử dụng

### ***Kết hợp Write-Through Cache với cache theo yêu cầu:***

**Bối cảnh:** Write-Through Cache có ưu điểm là dữ liệu luôn được cập nhật nhất quán với nguồn chính, nhưng có nhược điểm là tiêu tốn nhiều tài nguyên hệ thống. Cache theo yêu cầu có ưu điểm là tiết kiệm tài nguyên hệ thống, nhưng có nhược điểm là dữ liệu có thể không được cập nhật kịp thời.

#### **Phương pháp:**

o Sử dụng Write-Through Cache cho dữ liệu truy cập thường xuyên, ít thay đổi. o Sử dụng cache theo yêu cầu cho dữ liệu truy cập ít thường xuyên, thay đổi thường xuyên.

Ví dụ:

Lưu trữ dữ liệu sản phẩm trong Write-Through Cache: Dữ liệu sản phẩm được truy cập thường xuyên và ít thay đổi, do đó nên lưu trữ trong Write-Through Cache để đảm bảo truy cập nhanh chóng và nhất quán.

Lưu trữ dữ liệu lịch sử giao dịch trong cache theo yêu cầu: Dữ liệu lịch sử giao dịch được truy cập ít thường xuyên và thay đổi thường xuyên, do đó nên lưu trữ trong cache theo yêu cầu để tiết kiệm tài nguyên hệ thống.

## Chương 4. Các phương pháp cache

### 4.1. Write aside (Lazy caching)

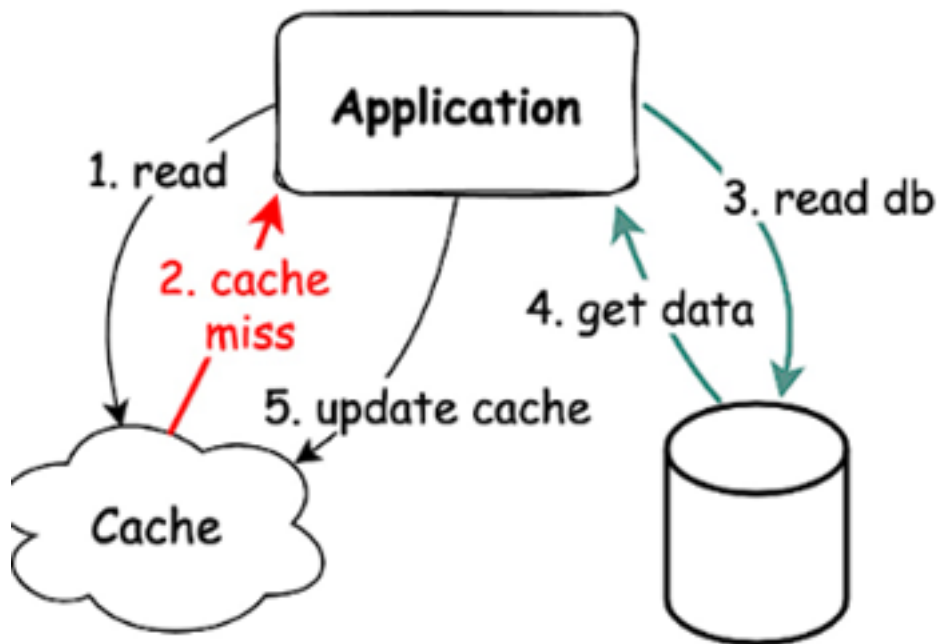
#### 4.1.1. Giới thiệu về caching

- Chiến lược lazy caching, còn được gọi là lazy population hoặc cache-aside, là một hình thức phổ biến của caching. Đây là chiến lược caching chỉ tải dữ liệu vào cache khi có yêu cầu truy cập. Chiến lược này có ưu điểm là tiết kiệm bộ nhớ và thời gian xử lý, vì chỉ những dữ liệu thực sự cần thiết mới được lưu trữ.
- Lazy caching giúp giữ kích thước cache quản lý được và chỉ lưu trữ những đối tượng mà ứng dụng thực sự yêu cầu, đồng thời tự động mở rộng cache khi có yêu cầu mới.
- Chiến lược lazy caching mang lại nhiều lợi ích, tuy nhiên không phải lúc nào cũng phù hợp để áp dụng. Dưới đây là một số trường hợp điển hình mà lazy caching phát huy hiệu quả:
  - Dữ liệu truy cập không thường xuyên:
    - Khi một đối tượng dữ liệu được truy cập không thường xuyên, việc lưu trữ nó trong cache có thể không mang lại lợi ích đáng kể. Lazy caching chỉ tải đối tượng vào cache khi có yêu cầu thực sự, giúp tiết kiệm bộ nhớ và thời gian xử lý cho những dữ liệu ít sử dụng.
  - Dữ liệu có chi phí truy cập cao:
    - Nếu việc truy cập dữ liệu từ cơ sở dữ liệu tốn kém về mặt tài nguyên (ví dụ: truy cập mạng, truy vấn phức tạp), lazy caching có thể giúp giảm tải cho cơ sở dữ liệu và cải thiện hiệu suất tổng thể của hệ thống.
  - Dữ liệu có khả năng thay đổi thấp:
    - Khi dữ liệu có khả năng thay đổi thấp, việc lưu trữ nó trong cache có thể mang lại hiệu quả cao hơn, vì dữ liệu trong cache có thể được sử dụng trong thời gian dài mà không cần cập nhật thường xuyên.
- Kích thước dữ liệu lớn:
  - Đối với những đối tượng dữ liệu có kích thước lớn, lazy caching giúp giảm thiểu lượng dữ liệu cần truyền tải giữa ứng dụng và cơ sở dữ liệu, cải thiện thời gian phản hồi và tiết kiệm băng thông.

- Hệ thống có nhiều người dùng:
  - Trong hệ thống có nhiều người dùng truy cập cùng lúc, lazy caching giúp giảm tải cho cơ sở dữ liệu và phân tán hiệu quả lưu lượng truy cập, đảm bảo hiệu suất ổn định cho tất cả người dùng.

---

## Read Strategy - Cache Aside



### 4.1.2. Cách hoạt động

#### 4.1.2.1. Tổng quan về luồng thực hiện

##### 4.1.2.1.1. Khi đọc dữ liệu:

- **Bước 1:** Ứng dụng trước tiên kiểm tra xem dữ liệu có tồn tại trong cache không.
- **Bước 2:** Nếu dữ liệu có trong cache (cache hit):
  - Lấy dữ liệu từ cache và trả về cho người dùng/quy trình.
- **Bước 3:** Nếu dữ liệu không có trong cache (cache miss):
  - Ứng dụng truy vấn trực tiếp đến nguồn dữ liệu (thường là cơ sở dữ liệu).
  - Sau khi nhận dữ liệu từ nguồn, ứng dụng lưu dữ liệu vào cache.
  - Trả về dữ liệu cho người dùng/quy trình.

##### 4.1.2.1.2. Khi ghi/cập nhật dữ liệu:

- **Bước 1:** Ứng dụng cập nhật trực tiếp vào nguồn dữ liệu (cơ sở dữ liệu).
- **Bước 2:** Sau khi cập nhật thành công:
  - Ứng dụng làm mất hiệu lực (invalidate) mục liên quan trong cache bằng cách xóa nó.

- **HOẶC** cập nhật mục trong cache với dữ liệu mới (tùy thuộc vào chiến lược).

#### **4.1.2.2. Chi tiết quá trình thực hiện**

##### **4.1.2.2.1. Bước xử lý yêu cầu đọc dữ liệu**

- Khi hệ thống nhận được yêu cầu đọc dữ liệu từ người dùng, đầu tiên nó kiểm tra xem dữ liệu đó có tồn tại trong cache không.
- Nếu dữ liệu đã tồn tại trong cache (cache hit), hệ thống sẽ trả về dữ liệu từ cache ngay lập tức, giúp đáp ứng nhanh chóng.
- Nếu dữ liệu không có trong cache (cache miss), hệ thống tiếp tục sang bước tiếp theo.

##### **4.1.2.2.2. Bước truy xuất dữ liệu từ nguồn chính**

- Khi xảy ra cache miss, hệ thống thực hiện truy vấn đến nguồn dữ liệu chính (thường là cơ sở dữ liệu).
- Hệ thống đọc dữ liệu cần thiết từ nguồn dữ liệu chính.
- Dữ liệu này được trả về cho người dùng/ứng dụng đã yêu cầu.

##### **4.1.2.2.3. Bước cập nhật cache**

- Sau khi lấy dữ liệu từ nguồn chính, hệ thống lưu trữ một bản sao của dữ liệu đó vào cache.
- Hệ thống gắn thông tin về thời gian hết hạn (TTL - Time To Live) cho dữ liệu trong cache (tùy thuộc vào cấu hình).
- Dữ liệu bây giờ đã sẵn sàng trong cache cho các yêu cầu đọc tiếp theo.

##### **4.1.2.2.4. Bước xử lý yêu cầu ghi dữ liệu**

- Khi có yêu cầu ghi (thêm, sửa, xóa) dữ liệu, hệ thống thực hiện thay đổi trực tiếp vào nguồn dữ liệu chính.
- **Điểm quan trọng:** Trong Lazy Caching, hệ thống không chủ động cập nhật cache khi có thay đổi dữ liệu.
- Cache không bị vô hiệu hóa hoặc cập nhật trực tiếp khi dữ liệu thay đổi ở nguồn chính.

##### **4.1.2.2.5. Bước quản lý sự nhất quán của dữ liệu**

- Để đảm bảo người dùng không bị trả về dữ liệu cũ, hệ thống sử dụng các cơ chế như:
  - Thiết lập thời gian sống (TTL) phù hợp cho các mục trong cache
  - Định kỳ làm mới toàn bộ cache hoặc một phần cache
  - Sử dụng cơ chế xóa cache có chọn lọc dựa trên các sự kiện xác định

##### **4.1.2.2.6. Bước xử lý khi cache đầy**

- Khi cache đạt đến giới hạn dung lượng được cấp phát, hệ thống cần quyết định những mục nào cần loại bỏ.

- Áp dụng các thuật toán loại bỏ (eviction policies) như:
  - LRU (Least Recently Used): Loại bỏ các mục ít được truy cập gần đây nhất
  - LFU (Least Frequently Used): Loại bỏ các mục ít được truy cập nhất
  - FIFO (First In First Out): Loại bỏ các mục được thêm vào cache sớm nhất

#### 4.1.3. Ưu và nhược điểm của Write aside (Lazy caching)

##### 4.1.3.1. Ưu điểm

- Đơn giản về mặt triển khai: Phương pháp này tương đối đơn giản để triển khai so với các cơ chế caching phức tạp khác, giúp giảm chi phí phát triển và bảo trì.
- Tối ưu hóa hoạt động đọc: Dữ liệu được đưa vào cache chỉ khi nó được yêu cầu, do đó cache chứa những dữ liệu thực sự cần thiết và được truy cập thường xuyên, tối ưu hóa không gian lưu trữ cache.
- Giảm độ trễ khi ghi: Vì dữ liệu được ghi trực tiếp vào hệ thống lưu trữ chính mà không phải đi qua cache, nên thao tác ghi có thể nhanh hơn trong nhiều trường hợp.
- Tính nhất quán cao: Vì dữ liệu được ghi trực tiếp vào bộ nhớ chính trước, nên tính nhất quán dữ liệu được đảm bảo tốt hơn so với một số phương pháp cache khác.
- Hiệu quả với dữ liệu truy cập không thường xuyên: Đối với dữ liệu ít khi được truy cập, write-aside tránh được việc lãng phí bộ nhớ cache để lưu trữ những dữ liệu này.

##### 4.1.3.2. Nhược điểm

- Cache miss ban đầu: Khi truy cập dữ liệu lần đầu, luôn xảy ra cache miss, dẫn đến độ trễ cao hơn cho lần truy cập đầu tiên vì phải đọc từ bộ nhớ chính.
- Hiệu suất đọc không ổn định: Có sự khác biệt lớn về thời gian phản hồi giữa cache hit và cache miss, có thể tạo ra trải nghiệm người dùng không ổn định.
- Không tối ưu cho dữ liệu thường xuyên cập nhật: Nếu dữ liệu thường xuyên bị thay đổi, cache có thể nhanh chóng trở nên lỗi thời, gây ra nhiều lần invalidation và reload.
- Khó khăn trong việc quản lý cache: Cần có cơ chế phức tạp để theo dõi và quản lý tính nhất quán giữa cache và bộ nhớ chính, đặc biệt trong môi trường phân tán.
- Không hiệu quả cho dữ liệu đọc-ghi cân bằng: Trong các ứng dụng có tỷ lệ đọc-ghi cân bằng, write-aside có thể không hiệu quả bằng các phương pháp khác như write-through.

#### 4.1.4. Ví dụ cài đặt

##### 4.1.4.1. Mục tiêu

- Cache dùng Redis để **tăng tốc độ đọc**.
- Tất cả thao tác **ghi chỉ thực hiện lên database (write-aside)**.
- Nếu dữ liệu được đọc mà **không có trong cache**, hệ thống sẽ lấy từ DB rồi **cập nhật vào cache**.

#### 4.1.4.2. Cấu trúc cơ bản

- Redis làm **cache**
- PostgreSQL hoặc bất kỳ DB nào làm **main memory**
- Spring Boot để xử lý logic
- Spring Data Redis để thao tác với cache

#### 4.1.4.3. Cài đặt

- Cài đặt Dependencies (Maven)

```
1 <dependencies>
2   <!-- Spring Data JPA -->
3   <dependency>
4     <groupId>org.springframework.boot</groupId>
5     <artifactId>spring-boot-starter-data-jpa</artifactId>
6   </dependency>
7
8   <!-- Redis -->
9   <dependency>
10    <groupId>org.springframework.boot</groupId>
11    <artifactId>spring-boot-starter-data-redis</artifactId>
12  </dependency>
13
14  <!-- PostgreSQL (hoặc database bạn dùng) -->
15  <dependency>
16    <groupId>org.postgresql</groupId>
17    <artifactId>postgresql</artifactId>
18  </dependency>
19 </dependencies>
```

- application.properties

```
1 # Redis
2 spring.redis.host=localhost
3 spring.redis.port=6379
4
5 # Database
6 spring.datasource.url=jdbc:postgresql://localhost:5432/demo
7 spring.datasource.username=postgres
8 spring.datasource.password=yourpassword
9 spring.jpa.hibernate.ddl-auto=update
```


- Entity



```

1  @Entity
2  public class Product {
3      @Id
4      private Long id;
5
6      private String name;
7      private Double price;
8
9      // Getters and setters
10 }

```


 Java

- Repository

```

1  public interface ProductRepository extends
    JpaRepository<Product, Long> {
2  }

```

 Java

- Service: Write-Aside Logic

```

1  @Service
2  public class ProductService {
3      @Autowired
4      private ProductRepository productRepository;
5
6      @Autowired
7      private RedisTemplate<String, Product> redisTemplate;
8
9      private final String CACHE_KEY_PREFIX = "product:";
10
11     public Product getProduct(Long id) {
12         String key = CACHE_KEY_PREFIX + id;
13         // 1. Try cache
14         Product product = redisTemplate.opsForValue().get(key);
15
16         if (product != null) {
17             System.out.println("Cache hit");
18             return product;
19         }
20
21         System.out.println("Cache miss");
22
23         // 2. Fallback to DB

```

 Java

```

24     Optional<Product> productOptional =
        productRepository.findById(id);
25     if (productOptional.isPresent()) {
26         product = productOptional.get();
27         redisTemplate.opsForValue().set(key, product); // update
            cache
28     }
29
30     return product;
31 }
32
33 public Product saveProduct(Product product) {
34     // 1. Write to DB (not cache)
35     Product saved = productRepository.save(product);
36
37     // 2. Optionally invalidate cache (hoặc bỏ qua để giữ nguyên
        write-aside)
38     redisTemplate.delete(CACHE_KEY_PREFIX + product.getId());
39
40     return saved;
41 }
42 }


```

- Controller

```

1  @RestController
2  @RequestMapping("/products")
3  public class ProductController {
4      @Autowired
5      private ProductService productService;
6
7      @GetMapping("/{id}")
8      public ResponseEntity<Product> getProduct(@PathVariable Long id) {
9          Product product = productService.getProduct(id);
10         return product != null ? ResponseEntity.ok(product) :
            ResponseEntity.notFound().build();
11     }
12
13     @PostMapping
14     public ResponseEntity<Product> create(@RequestBody Product
        product) {
15         Product saved = productService.saveProduct(product);

```

 Java

```
16         return ResponseEntity.ok(saved);  
17     }  
18 }
```

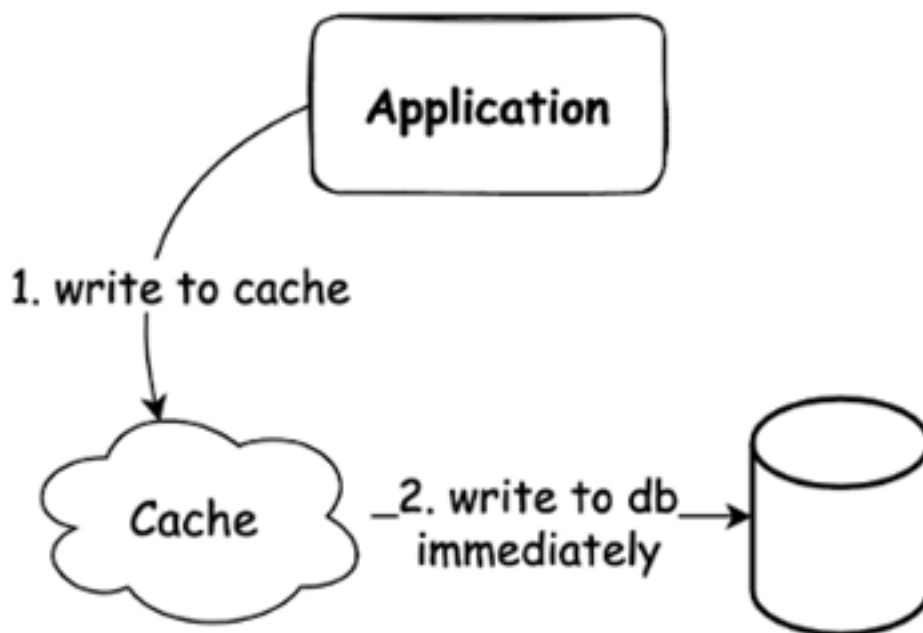
## 4.2. Write-Through

### 4.2.1. Giới thiệu Write-Through

Write-Through là một phương pháp lưu trữ trong đó dữ liệu được ghi đồng thời vào bộ nhớ đệm (cache) và vị trí bộ nhớ chính tương ứng.

Với bộ nhớ đệm write-through, đơn vị xử lý trung tâm (CPU) của máy tính ghi dữ liệu vào bộ nhớ đệm cục bộ của bộ xử lý và tạm dừng việc thực thi ứng dụng tương ứng cho đến khi cùng một dữ liệu cũng được ghi thành công vào tài nguyên lưu trữ liên quan, thường là bộ nhớ chính (RAM) hoặc ổ đĩa lưu trữ. Bộ xử lý duy trì khả năng truy cập tốc độ cao đến dữ liệu được lưu trong bộ nhớ đệm, đồng thời đảm bảo sự nhất quán hoàn toàn giữa bộ nhớ đệm và kho lưu trữ dữ liệu.

### Write Strategy - Write Through



### 4.2.2. Hoạt động của bộ nhớ đệm write-through

- Khái niệm về bộ nhớ đệm đã được thiết lập vững chắc trong thiết kế máy tính như một phương tiện để tăng tốc hiệu suất của bộ xử lý và máy tính bằng cách đặt dữ liệu thường xuyên sử dụng trong một lượng nhỏ bộ nhớ cực kỳ nhanh - bộ nhớ đệm - được đặt liền kề với bộ xử lý. Điều này cho phép bộ xử lý tìm dữ liệu thường dùng nhanh hơn nhiều so với việc truy cập cùng một dữ liệu từ đĩa hoặc thậm chí từ bộ

nhớ chính nhanh hơn. Các bộ xử lý hiện đại thường làm việc với nhiều cấp độ bộ nhớ đệm, chẳng hạn như bộ nhớ đệm cấp một (L1) hoặc bộ nhớ đệm cấp hai (L2).

- Khi bộ xử lý ghi hoặc xuất dữ liệu, dữ liệu đó trước tiên được đặt trong bộ nhớ đệm. Nội dung bộ nhớ đệm sau đó được ghi vào bộ nhớ chính hoặc đĩa sử dụng một trong ba thuật toán hoặc chính sách chính sau đây:
- **Bộ nhớ đệm write-through:** Với chính sách bộ nhớ đệm write-through, bộ xử lý của hệ thống ghi dữ liệu vào bộ nhớ đệm trước, sau đó ngay lập tức sao chép dữ liệu bộ nhớ đệm mới đó vào bộ nhớ hoặc đĩa tương ứng. Ứng dụng đang làm việc để tạo ra dữ liệu này tạm dừng thực thi cho đến khi việc ghi dữ liệu mới vào bộ nhớ hoàn tất. Bộ nhớ đệm write-through đảm bảo dữ liệu luôn nhất quán giữa bộ nhớ đệm và bộ nhớ lưu trữ, mặc dù hiệu suất ứng dụng có thể bị ảnh hưởng một chút vì ứng dụng phải chờ các hoạt động nhập/xuất lâu hơn vào bộ nhớ hoặc thậm chí vào đĩa.

#### 4.2.3. Mục đích bộ nhớ đệm write-through

- Bộ nhớ đệm write-through là một kỹ thuật hoặc chính sách bộ nhớ đệm kiểm soát cách bộ xử lý và bộ nhớ đệm cục bộ của nó tương tác với các tài sản lưu trữ chính khác trong máy tính, chẳng hạn như RAM hoặc đĩa - cho dù là ổ đĩa thể rắn (SSD) hoặc ổ đĩa cứng truyền thống (HDD) - và một số ứng dụng doanh nghiệp tập trung vào lưu trữ như Structured Query Language (SQL).
- Bộ nhớ đệm write-through, như tên gọi của nó, ghi dữ liệu vào bộ nhớ đệm cục bộ của bộ xử lý trước tiên và sau đó ngay lập tức ghi dữ liệu được lưu trữ đó đến mục tiêu lưu trữ cuối cùng trong bộ nhớ hoặc đĩa. Việc thực thi ứng dụng tạm dừng cho đến khi dữ liệu được ghi thành công vào tài sản lưu trữ cuối cùng.
- Mục đích của bộ nhớ đệm write-through là để đạt được sự nhất quán dữ liệu giữa bộ nhớ đệm của bộ xử lý và bộ nhớ lưu trữ ứng dụng. Bộ nhớ đệm write-through đảm bảo dữ liệu trong bộ nhớ đệm và bộ nhớ lưu trữ luôn giống hệt nhau, vì vậy không có khả năng mất dữ liệu hoặc hỏng nếu ứng dụng bị sự cố hoặc hệ thống máy tính bị lỗi trước khi bộ nhớ đệm được ghi, điều này có thể xảy ra với bộ nhớ đệm write-back. Bộ nhớ đệm write-through thường là kỹ thuật ưa thích cho các tác vụ máy tính quan trọng không thể chấp nhận rủi ro mất dữ liệu.
- Tuy nhiên, bộ nhớ đệm write-through giữ việc thực thi ứng dụng cho đến khi dữ liệu mới trong bộ nhớ đệm được cam kết vào bộ nhớ lưu trữ. Điều này có thể áp đặt một hình phạt nhỏ đối với hiệu suất ứng dụng hiệu quả hoặc rõ ràng.

#### 4.2.4. Ưu và nhược điểm của bộ nhớ đệm write-through

- Như một khái niệm chung, bộ nhớ đệm bộ xử lý thường giúp nâng cao hiệu suất ứng dụng bằng cách cho phép bộ xử lý truy cập dữ liệu gần đây từ bộ nhớ đệm nhanh hơn nhiều so với nó có thể truy cập cùng một dữ liệu từ các tài nguyên lưu trữ khác như bộ nhớ chính hoặc đĩa. Tuy nhiên, để bộ nhớ đệm hữu ích, ứng dụng cần bộ xử

lý đọc cùng một dữ liệu được lưu trong bộ nhớ đệm thường xuyên - nếu không thì không cần thiết phải có bộ nhớ đệm.

- Bộ nhớ đệm write-through cung cấp lợi ích hiệu suất bộ xử lý này bằng cách ghi dữ liệu vào bộ nhớ đệm trước. Tuy nhiên, dữ liệu mới được đặt vào bộ nhớ đệm cũng được cam kết hoặc sao chép đến vị trí tương ứng trong bộ nhớ chính hoặc đĩa trước khi việc thực thi ứng dụng được phép tiếp tục. Điều này đảm bảo dữ liệu trong bộ nhớ đệm và bộ nhớ lưu trữ luôn nhất quán và không còn dữ liệu bị mất hoặc hỏng nếu ứng dụng bị lỗi vì bất kỳ lý do gì. Bộ nhớ đệm write-through thường là một kỹ thuật ưa thích cho các ứng dụng quan trọng không thể chấp nhận rủi ro mất dữ liệu.
- Tuy nhiên, quá trình write-through phải tạm dừng việc thực thi ứng dụng cho đến khi dữ liệu được cam kết đầy đủ vào bộ nhớ lưu trữ. Mặc dù điều này có thể chỉ mất vài mili giây, việc lưu trữ đệm thường xuyên dẫn đến các lần tạm dừng thường xuyên, và điều này có thể làm giảm hiệu suất rõ ràng của ứng dụng. Người dùng thường không thể nhận thấy bất kỳ tác động nào của các chính sách lưu trữ đệm đối với hiệu suất ứng dụng, nhưng các ứng dụng tập trung vào hiệu suất có thể không phù hợp với bộ nhớ đệm write-through.

#### 4.2.5. Ví dụ triển khai

##### 4.2.5.1. Cấu trúc

- **Redis** là Cache
- **PostgreSQL/MySQL** là Database
- **Spring Boot** làm backend logic

##### 4.2.5.2. Maven Dependencies (giống write-aside)

1 <!-- Thêm các dependency như đã trình bày ở write-aside -->

XML

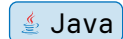
##### 4.2.5.3. Entity

```
1 @Entity
2 public class Product {
3     @Id
4     private Long id;
5
6     private String name;
7     private Double price;
8
9     // Getters và setters
10 }
```

Java

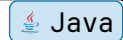
#### 4.2.5.4. Repository

```
1 public interface ProductRepository extends
2   JpaRepository<Product, Long> {
```



#### 4.2.5.5. Service – Write-Through Cache Logic

```
1 @Service
2 public class ProductService {
3     @Autowired
4     private ProductRepository productRepository;
5
6     @Autowired
7     private RedisTemplate<String, Product> redisTemplate;
8
9     private final String CACHE_KEY_PREFIX = "product:";
10
11     public Product getProduct(Long id) {
12         String key = CACHE_KEY_PREFIX + id;
13
14         Product product = redisTemplate.opsForValue().get(key);
15         if (product != null) {
16             System.out.println("Cache hit");
17             return product;
18         }
19
20         System.out.println("Cache miss");
21         Optional<Product> optionalProduct =
22             productRepository.findById(id);
23         if (optionalProduct.isPresent()) {
24             product = optionalProduct.get();
25             redisTemplate.opsForValue().set(key, product); // Cập nhật
26             cache
27         }
28
29         return product;
30     }
31
32     public Product saveProduct(Product product) {
33         // 1. Ghi vào DB
34         Product saved = productRepository.save(product);
```



```

34         // 2. Ghi vào cache ngay lập tức (write-through)
35         redisTemplate.opsForValue().set(CACHE_KEY_PREFIX +
36         saved.getId(), saved);
37
38         return saved;
39     }
40
41     public Product updateProduct(Product product) {
42         return saveProduct(product); // Ghi lại DB và cache
43     }
44
45     public void deleteProduct(Long id) {
46         // Xoá DB
47         productRepository.deleteById(id);
48         // Xoá cache
49         redisTemplate.delete(CACHE_KEY_PREFIX + id);
50     }

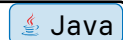
```

#### 4.2.5.6. Controller

```

1  @RestController
2  @RequestMapping("/products")
3  public class ProductController {
4      @Autowired
5      private ProductService productService;
6
7      @GetMapping("/{id}")
8      public ResponseEntity<Product> getProduct(@PathVariable Long id) {
9          Product product = productService.getProduct(id);
10         return product != null ? ResponseEntity.ok(product) :
11         ResponseEntity.notFound().build();
12     }
13
14     @PostMapping
15     public ResponseEntity<Product> create(@RequestBody Product
16     product) {
17         Product saved = productService.saveProduct(product);
18         return ResponseEntity.ok(saved);
19     }
20
21     @PutMapping

```



```

20     public ResponseEntity<Product> update(@RequestBody Product
      product) {
21         Product updated = productService.updateProduct(product);
22         return ResponseEntity.ok(updated);
23     }
24
25     @DeleteMapping("/{id}")
26     public ResponseEntity<Void> delete(@PathVariable Long id) {
27         productService.deleteProduct(id);
28         return ResponseEntity.ok().build();
29     }
30 }

```

## 4.3. Write behind (Write back)

### 4.3.1. Giới thiệu Write-Behind Caching

- Có một thách thức lớn với bộ nhớ đệm write-through (write-through): độ trễ cao của các thao tác ghi vì dữ liệu cần được cập nhật đồng bộ ở cả hai nơi, nghĩa là trước tiên trong bộ nhớ đệm và sau đó trong cơ sở dữ liệu chính. Một giải pháp để cải thiện hiệu suất ghi là sử dụng bộ nhớ đệm write-behind (write-behind/write-back).
- Ý tưởng của bộ nhớ đệm write-behind tương tự như bộ nhớ đệm write-through, nhưng có một sự khác biệt đáng kể: dữ liệu được ghi vào bộ nhớ đệm được cập nhật bất đồng bộ trong cơ sở dữ liệu chính. Nói cách khác, ứng dụng sẽ ghi vào bộ nhớ đệm trước, và bộ nhớ đệm sẽ ghi vào cơ sở dữ liệu sau một khoảng thời gian trễ.
- Hãy hiểu điều này từ một góc độ khác! Trong bộ nhớ đệm write-through, khi nhiều yêu cầu ghi đến trong một khoảng thời gian ngắn, cơ sở dữ liệu có thể dễ dàng trở thành nút thắt cổ chai. Vì vậy, bộ nhớ đệm write-behind cho phép ghi vào bộ nhớ đệm nhanh hơn với tính nhất quán dữ liệu cuối cùng giữa bộ nhớ đệm và cơ sở dữ liệu. Trong khi đó, bất kỳ thao tác đọc nào từ bộ nhớ đệm vẫn sẽ nhận được dữ liệu mới nhất.
- Do đó, ứng dụng không cần phải đợi cập nhật cơ sở dữ liệu, tức là ứng dụng chỉ ghi dữ liệu vào bộ nhớ đệm, và bộ nhớ đệm xác nhận **ghi ngay lập tức**. Bộ nhớ đệm theo dõi các thay đổi được thực hiện và ghi chúng trở lại cơ sở dữ liệu chính vào thời điểm sau (có thể trong thời gian ít bận rộn hơn). Cập nhật bất đồng bộ này sẽ giảm độ trễ của thao tác ghi.
- Bây giờ câu hỏi đặt ra là: Làm thế nào chúng ta có thể thực hiện cập nhật bất đồng bộ từ bộ nhớ đệm đến cơ sở dữ liệu? Một ý tưởng là sử dụng độ trễ dựa trên thời gian, trong đó bộ nhớ đệm đợi một khoảng thời gian định trước trước khi thực hiện cập nhật vào cơ sở dữ liệu. Một ý tưởng khác là sử dụng độ trễ dựa trên số lượng



mục nhập, trong đó bộ nhớ đệm đợi cho đến khi tích lũy một số lượng mục dữ liệu mới nhất định trước khi cập nhật cơ sở dữ liệu.

alt text

#### **4.3.2. Ưu và nhược điểm của mô hình write-behind caching**

##### **4.3.2.1. Ưu điểm**

- Bộ nhớ đệm write-behind cải thiện hiệu suất của các thao tác ghi, vì vậy đây là một lựa chọn tốt khi xử lý các khối lượng công việc nặng về ghi. Ngoài ra, kết hợp với mô hình write-through (read-through), write-behind caching cũng phù hợp cho các khối lượng công việc hỗn hợp liên quan đến cả thao tác đọc và ghi. Hãy khám phá và suy nghĩ!
- Bộ nhớ đệm xác nhận ghi ngay lập tức và trì hoãn cập nhật cơ sở dữ liệu. Vì vậy, điều này sẽ giảm áp lực lên bộ nhớ đệm. Ngay cả khi cơ sở dữ liệu gặp thời gian chết hoặc lỗi, ứng dụng vẫn có thể hoạt động và phục vụ các yêu cầu đọc và ghi từ bộ nhớ đệm. Các cập nhật chưa được ghi vào cơ sở dữ liệu được xếp hàng đợi trong bộ nhớ đệm và có thể đồng bộ hóa với cơ sở dữ liệu khi nó phục hồi. Điều này nâng cao tính khả dụng tổng thể của hệ thống và giảm thiểu tác động của các lỗi cơ sở dữ liệu.
- Bộ nhớ đệm write-behind cũng giảm tải cho cơ sở dữ liệu, vì cơ sở dữ liệu được cập nhật ít thường xuyên hơn.
- Bây giờ câu hỏi đặt ra là: Chúng ta có thể giảm thêm tải cho cơ sở dữ liệu không?

##### **4.3.2.2. Nhược điểm**

- Độ trễ giữa ghi bộ nhớ đệm và ghi cơ sở dữ liệu tạo ra một khoảng thời gian mà trong đó dữ liệu trong bộ nhớ đệm chưa được phản ánh trong cơ sở dữ liệu. Trong khoảng thời gian này, bộ nhớ đệm lưu trữ dữ liệu mới nhất và cơ sở dữ liệu có thể lưu trữ dữ liệu cũ, tức là bộ nhớ đệm có thể đi trước cơ sở dữ liệu về mặt tính nhất quán dữ liệu. Vì vậy, chúng ta nên sử dụng bộ nhớ đệm write-behind khi ứng dụng được thiết kế để chấp nhận dữ liệu không nhất quán trong thời gian trễ.
- Nếu có lỗi bộ nhớ đệm hoặc sự cố hệ thống, dữ liệu được ghi gần đây trong bộ nhớ đệm có thể bị mất vĩnh viễn. Điều này có thể ảnh hưởng đến hoạt động tổng thể của ứng dụng.
- Trong bộ nhớ đệm write-behind, nếu xảy ra lỗi trong quá trình ghi trễ vào cơ sở dữ liệu sau khi thao tác ghi bộ nhớ đệm đã được xác nhận với ứng dụng, sẽ có khả năng mất dữ liệu hoặc dữ liệu không nhất quán.
- Làm thế nào để xử lý tình huống này? Một ý tưởng là thực hiện cơ chế thử lại hoặc sử dụng ghi trễ với thời gian chờ thích hợp.

#### **4.3.3. Các phương pháp để giảm tải cho cơ sở dữ liệu**

- Ý tưởng là sử dụng các chiến lược này kết hợp với phương pháp write-behind caching để tối ưu hóa quá trình ghi và giảm tải cho cơ sở dữ liệu bằng cách xử lý các đợt tăng đột biến ghi một cách duyên dáng hơn.

##### **4.3.3.1. Sử dụng giới hạn tỷ lệ (rate limiting)**

- Khi có nhiều yêu cầu ghi đến cùng một lúc, nó có thể làm quá tải cơ sở dữ liệu. Để tránh điều này, chúng ta có thể sử dụng giới hạn tỷ lệ trong bộ nhớ đệm write-behind để đặt giới hạn về số lượng yêu cầu ghi mà cơ sở dữ liệu có thể xử lý mỗi giây hoặc mỗi phút.
- Điều này sẽ phân tán khối lượng công việc ghi trong một khoảng thời gian dài hơn, đảm bảo luồng ghi đều đặn vào cơ sở dữ liệu thay vì đột biến đột ngột trong thời kỳ cao điểm. Nó sẽ cho cơ sở dữ liệu đủ thời gian để theo kịp và xử lý các yêu cầu ghi mà không bị quá tải.

##### **4.3.3.2. Sử dụng kỹ thuật phân lô (batching) và hợp nhất (coalescing)**

- Chúng ta có thể sử dụng kỹ thuật phân lô và hợp nhất trong bộ nhớ đệm write-behind để giảm số lượng yêu cầu ghi. Phân lô kết hợp nhiều thao tác ghi thành một lần ghi duy nhất, trong khi hợp nhất củng cố nhiều bản cập nhật trên cùng một dữ liệu thành một bản cập nhật duy nhất.
- Điều này có nghĩa là thay vì ghi ngay từng thay đổi vào cơ sở dữ liệu, bộ nhớ đệm có thể nhóm chúng lại và ghi chúng như một thao tác duy nhất. Kết quả là, điều này sẽ giảm tổng số lần ghi vào cơ sở dữ liệu, hiệu quả giảm tải cho cơ sở dữ liệu. Điều tốt là: Nó cũng sẽ tiết kiệm chi phí khi nhà cung cấp cơ sở dữ liệu tính phí dựa trên số lượng yêu cầu được thực hiện.

##### **4.3.3.3. Sử dụng chuyển dịch thời gian (time shifting)**

- Cơ sở dữ liệu có thể trải qua “giờ cao điểm” khi nhiều dữ liệu đang được ghi hoặc sửa đổi đồng thời. Vì vậy, chuyển dịch thời gian là một ý tưởng để chiến lược di chuyển quá trình ghi dữ liệu đến thời điểm ít bận rộn hơn hoặc khoảng thời gian khác ngoài thời gian sử dụng cao điểm. Điều này sẽ cho phép hệ thống tránh bị quá tải trong thời kỳ tranh chấp cao.

#### **4.3.4. So sánh write-behind với write-through cache**

- Quyết định giữa chiến lược bộ nhớ đệm write-through và write-behind caching liên quan đến một trong những đánh đổi quan trọng: tính nhất quán dữ liệu so với hiệu suất ghi. Đây là so sánh quan trọng giữa cả hai phương pháp:
  - Write-through duy trì tính nhất quán giữa bộ nhớ đệm và cơ sở dữ liệu. Mặt khác, write-behind caching có thể dẫn đến sự không nhất quán tạm thời giữa cơ sở dữ liệu và bộ nhớ đệm.

- ▶ Vì dữ liệu được ghi ngay lập tức vào cả bộ nhớ đệm và cơ sở dữ liệu, bộ nhớ đệm write-through cung cấp độ bền dữ liệu mạnh mẽ. Trong trường hợp lỗi hệ thống, dữ liệu đã cập nhật sẽ có sẵn trong cơ sở dữ liệu. Mặt khác, có nguy cơ mất dữ liệu tiềm ẩn trong bộ nhớ đệm write-behind, đặc biệt là trong trường hợp mất điện đột ngột hoặc lỗi bộ nhớ đệm trước khi dữ liệu được cập nhật trong cơ sở dữ liệu.
- ▶ Write-through tạo ra độ trễ cao cho các thao tác ghi vì dữ liệu cần được ghi vào cả bộ nhớ đệm và cơ sở dữ liệu. Mặt khác, bộ nhớ đệm write-behind thường cung cấp độ trễ thấp cho các thao tác ghi so với write-through vì dữ liệu được ghi đầu tiên vào bộ nhớ đệm và ứng dụng có thể tiếp tục mà không có độ trễ ghi ngay lập tức.
- ▶ write-through phù hợp với dữ liệu nhạy cảm cao hoặc kịch bản mà tính nhất quán và toàn vẹn dữ liệu là ưu tiên hàng đầu. Mặt khác, bộ nhớ đệm write-behind phù hợp cho dữ liệu không quan trọng hoặc kịch bản mà hiệu suất ghi cho khối lượng công việc nặng về ghi là quan trọng.

#### 4.3.5. Ví dụ triển khai

##### 4.3.5.1. Mục tiêu:

- Ghi dữ liệu vào cache trước, sau đó ghi về database sau (delayed write).
- Ghi xuống database có thể thực hiện sau vài giây hoặc theo batch.
- Đọc vẫn ưu tiên cache.

##### 4.3.5.2. Cách triển khai

##### 4.3.5.3. Ý tưởng:

- Khi ghi → lưu vào cache (Redis hoặc Map)
- Có một scheduler chạy nền, mỗi X giây lấy dữ liệu trong cache → ghi về DB

##### 4.3.5.4. Maven dependencies (giống các ví dụ trước)

```
1 <!-- Spring Boot Data JPA, Redis, PostgreSQL -->
```

XML

##### 4.3.5.5. Entity & Repository

```
1 @Entity
2 public class Product {
3     @Id
4     private Long id;
5
6     private String name;
7     private Double price;
8
9     // Getters/setters
```

Java

```

10 }
11
12 public interface ProductRepository extends JpaRepository<Product,
13     Long> {

```

#### 4.3.5.6. Cache Store (dùng ConcurrentHashMap hoặc Redis)

```

1  @Component
2  public class InMemoryCache {
3      private final Map<Long, Product> writeBehindBuffer = new
4          ConcurrentHashMap<>();
5
6      public void put(Product product) {
7          writeBehindBuffer.put(product.getId(), product);
8      }
9
10     public Product get(Long id) {
11         return writeBehindBuffer.get(id);
12     }
13
14     public boolean contains(Long id) {
15         return writeBehindBuffer.containsKey(id);
16     }
17
18     public Collection<Product> flush() {
19         List<Product> products = new
20             ArrayList<>(writeBehindBuffer.values());
21         writeBehindBuffer.clear();
22         return products;
23     }
24
25     public Map<Long, Product> getAll() {
26         return writeBehindBuffer;
27     }

```

#### 4.3.5.7. Service: Write-Behind Logic

```

1  @Service
2  public class ProductService {
3      @Autowired
4      private ProductRepository productRepository;

```

```

5
6     @Autowired
7     private InMemoryCache cache;
8
9     public Product getProduct(Long id) {
10         if (cache.contains(id)) {
11             System.out.println("Cache hit");
12             return cache.get(id);
13         }
14
15         System.out.println("Cache miss, reading DB");
16         Optional<Product> product = productRepository.findById(id);
17         product.ifPresent(p → cache.put(p)); // Cache lại nếu lấy từ DB
18         return product.orElse(null);
19     }
20
21     public Product saveProduct(Product product) {
22         // ⚡ Ghi vào cache thôi, chưa ghi DB
23         cache.put(product);
24         return product;
25     }
26
27     public void deleteProduct(Long id) {
28         cache.getAll().remove(id);
29         productRepository.deleteById(id);
30     }
31 }

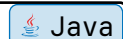
```

#### 4.3.5.8. Scheduled Flush Job (ghi DB định kỳ)

```

1  @Component
2  public class WriteBehindScheduler {
3      @Autowired
4      private InMemoryCache cache;
5
6      @Autowired
7      private ProductRepository repository;
8
9      @Scheduled(fixedRate = 10000) // 10 giây
10     public void flushCacheToDatabase() {
11         Collection<Product> pendingWrites = cache.flush();

```



```

12
13     if (!pendingWrites.isEmpty()) {
14         System.out.println("Flushing to DB: " +
15             pendingWrites.size() + " records");
16         repository.saveAll(pendingWrites);
17     }
18 }

```

Đừng quên thêm `@EnableScheduling` vào class `@SpringBootApplication`

#### 4.3.5.9. Controller

```

1  @RestController
2  @RequestMapping("/products")
3  public class ProductController {
4      @Autowired
5      private ProductService productService;
6
7      @GetMapping("/{id}")
8      public ResponseEntity<Product> get(@PathVariable Long id) {
9          Product p = productService.getProduct(id);
10         return p != null ? ResponseEntity.ok(p) :
11             ResponseEntity.notFound().build();
12     }
13
14     @PostMapping
15     public ResponseEntity<Product> create(@RequestBody Product p) {
16         return ResponseEntity.ok(productService.saveProduct(p));
17     }

```

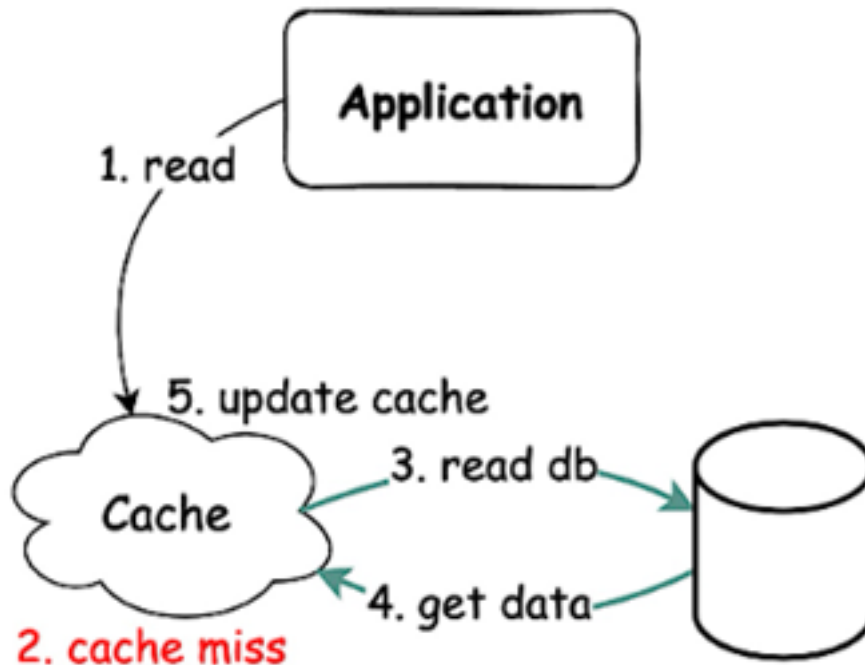
### 4.4. Read through

#### 4.4.1. Giới thiệu về Read-Through

- Khác với mô hình cache-aside (bộ nhớ đệm phụ trợ) yêu cầu logic ứng dụng để tìm nạp và đưa dữ liệu vào bộ nhớ đệm, bộ nhớ đệm Read-Through quản lý việc truy xuất dữ liệu từ cơ sở dữ liệu thay mặt cho ứng dụng. Trong trường hợp này, ứng dụng sẽ tương tác với hệ thống bộ nhớ đệm đóng vai trò trung gian giữa ứng dụng và cơ sở dữ liệu.
- Một trong những lợi ích quan trọng của mô hình read-through là đơn giản hóa mã ứng dụng. Nó chuyển trách nhiệm xử lý các trường hợp nhớ đệm trượt (cache miss) và truy xuất dữ liệu sang tầng bộ nhớ đệm. Nhờ vậy, ứng dụng chỉ tương tác với bộ

nhớ đệm như thể đó là nguồn dữ liệu chính.

## Read Strategy - Read Through



### 4.4.2. Cách hoạt động của mô hình read-through

- Khi ứng dụng yêu cầu dữ liệu từ bộ nhớ đệm, bộ nhớ đệm sẽ kiểm tra xem dữ liệu có hiện diện không (cache hit). Nếu có, dữ liệu được lưu trong bộ nhớ đệm sẽ được trả về trực tiếp cho ứng dụng. Nếu dữ liệu không có (cache miss), bộ nhớ đệm sẽ đóng vai trò trung gian và lấy dữ liệu từ cơ sở dữ liệu. Sau đó, bộ nhớ đệm sẽ lưu trữ dữ liệu đã truy xuất và trả lại cho ứng dụng như kết quả của yêu cầu đọc ban đầu. Điều này cho phép các yêu cầu đọc tiếp theo đối với cùng một dữ liệu được phục vụ hiệu quả từ bộ nhớ đệm với độ trễ thấp.
- Đối với việc ghi dữ liệu trong bộ nhớ đệm read-through, chúng ta có thể áp dụng những chiến lược sau tùy theo yêu cầu: chiến lược Write-around, Write-through và Write-back (hoặc write-behind). Lựa chọn phương pháp ghi tốt nhất phụ thuộc vào sự đánh đổi về tính nhất quán của dữ liệu và hiệu suất ghi.

### 4.4.3. Ưu và nhược điểm của Read-through

#### 4.4.3.1. Ưu điểm

- Bộ nhớ đệm Read-through giảm đáng kể độ trễ đọc đối với dữ liệu được truy cập thường xuyên, đặc biệt khi nguồn dữ liệu gốc nằm ở xa (trung tâm dữ liệu cách xa về mặt địa lý). Đó là lý do tại sao nó được sử dụng trong hệ thống có mẫu truy cập dữ liệu thiên về đọc.

- Với mô hình write-through, bộ nhớ đệm read-through cung cấp cách tự động đảm bảo tính nhất quán của dữ liệu. Mặt khác, read-through với mô hình write-around không cần biết đầy đủ về tất cả dữ liệu có thể được yêu cầu trong tương lai. Thay vào đó, nó có thể lưu dữ liệu vào bộ nhớ đệm khi có yêu cầu đọc (tải lười biếng - lazy loading). Điều này cho phép hệ thống tải dữ liệu vào bộ nhớ đệm theo nhu cầu và tránh làm đầy bộ nhớ đệm với dữ liệu không được yêu cầu.
- Vì bộ nhớ đệm read-through phục vụ dữ liệu từ bộ nhớ đệm bất cứ khi nào có thể, nó giảm số lượng yêu cầu trực tiếp đến nguồn dữ liệu cơ bản.
- Chúng ta có thể cấu hình bộ nhớ đệm read-through để tự động tải lại dữ liệu từ cơ sở dữ liệu khi nó hết hạn hoặc khi dữ liệu tương ứng thay đổi trong cơ sở dữ liệu. Vì vậy, trong tình huống lưu lượng truy cập cao, có thể có ít khả năng xảy ra trượt bộ nhớ đệm cho các mục bộ nhớ đệm này.
- Nó đơn giản hóa mã ứng dụng và trừu tượng hóa độ phức tạp của quản lý bộ nhớ đệm và truy xuất dữ liệu. Vì vậy, nhà phát triển không cần phải triển khai các logic phức tạp để xử lý bộ nhớ đệm theo cách thủ công.
- Trong trường hợp một nút bị lỗi, chúng ta có thể dễ dàng thay thế nút đó bằng một nút trống mới, vì vậy ứng dụng có thể tiếp tục hoạt động. Nhưng độ trễ sẽ tăng lên lúc đầu vì sẽ có khả năng cao xảy ra trượt bộ nhớ đệm đối với một số truy vấn đọc ban đầu. Một giải pháp cho vấn đề độ trễ này là làm nóng bộ nhớ đệm bằng cách đưa vào dữ liệu dự kiến sẽ được yêu cầu thường xuyên trong tương lai gần.

#### **4.4.3.2. Nhược điểm**

- Bất cứ khi nào có yêu cầu đọc mới hoặc dữ liệu được yêu cầu lần đầu tiên hoặc sau khi TTL (thời gian sống) kết thúc, nó sẽ luôn dẫn đến trượt bộ nhớ đệm. Ý tưởng rất đơn giản: Trong tất cả các tình huống như vậy, yêu cầu đọc sẽ đi đến cơ sở dữ liệu. Điều này sẽ tạo thêm độ trễ: Ba chuyển đi mạng
  - Kiểm tra bộ nhớ đệm
  - Truy xuất từ cơ sở dữ liệu
  - Thêm dữ liệu vào bộ nhớ đệm.
- Với bộ nhớ đệm write-through, bộ nhớ đệm read-through có thể dẫn đến việc lưu vào bộ nhớ đệm dữ liệu ít được truy cập hoặc không bao giờ được truy cập lại. Tình huống này có thể tiêu tốn không gian bộ nhớ đệm quý giá lẽ ra có thể được sử dụng cho dữ liệu phù hợp hơn và thường xuyên được truy cập.
- Giống như cache-aside, dữ liệu cũng có thể trở nên không nhất quán giữa bộ nhớ đệm và cơ sở dữ liệu. Ví dụ: nếu có thay đổi trong cơ sở dữ liệu và khóa bộ nhớ đệm chưa hết hạn, nó có thể trả về dữ liệu cũ cho ứng dụng. Một giải pháp là sử dụng các chiến lược ghi phù hợp như bộ nhớ đệm write-through và cấu hình chiến lược loại bỏ phù hợp để đảm bảo tính nhất quán của dữ liệu.



- Như đã thảo luận ở trên, chúng ta có thể tự động tải lại dữ liệu trong bộ nhớ đệm từ cơ sở dữ liệu khi nó hết hạn. Vấn đề là: Nhiều dữ liệu trong bộ nhớ đệm hết hạn đồng thời có thể dẫn đến nhiều yêu cầu cơ sở dữ liệu (cache stampede - tình trạng tràn ngập bộ nhớ đệm). Những yêu cầu đồng thời này gây ra đột biến tải trên cơ sở dữ liệu.

#### 4.4.4. Read-Through so với Cache-Aside (Lazy cache)

- Mặc dù bộ nhớ đệm read-through và cache-aside trông rất giống nhau, nhưng có một số khác biệt lớn:
  - Bộ nhớ đệm Read-through đặt logic bộ nhớ đệm trong chính hệ thống bộ nhớ đệm và giảm gánh nặng cho ứng dụng trong việc quản lý logic bộ nhớ đệm. Cache-aside đặt logic bộ nhớ đệm trong ứng dụng. Ứng dụng quyết định khi nào lưu trữ dữ liệu, khi nào loại bỏ dữ liệu và cách xử lý các trường hợp trượt bộ nhớ đệm. Điều này sẽ cung cấp nhiều quyền kiểm soát hơn cho ứng dụng.
  - Trong bộ nhớ đệm read-through, ứng dụng tương tác với bộ nhớ đệm như thể đó là nguồn dữ liệu chính. Ứng dụng không biết liệu dữ liệu đã được lưu trữ hay chưa, vì bộ nhớ đệm quản lý việc này một cách nội bộ. Trong cache-aside, ứng dụng phải kiểm tra rõ ràng bộ nhớ đệm trước khi truy cập nguồn dữ liệu.
  - Trong mô hình cache aside, nếu ứng dụng không xử lý cập nhật bộ nhớ đệm đúng cách, có thể xảy ra khả năng dữ liệu không nhất quán. Vì vậy, nhà phát triển ứng dụng có trách nhiệm viết logic cho việc vô hiệu hóa bộ nhớ đệm hoặc cập nhật dữ liệu đã lưu trong bộ nhớ đệm một cách chính xác. Mặt khác, khi dữ liệu được cập nhật trong nguồn dữ liệu, bộ nhớ đệm read-through có thể được cấu hình dễ dàng để tự động cập nhật hoặc vô hiệu hóa mục đã lưu trong bộ nhớ đệm tương ứng.

#### 4.4.5. Ví dụ triển khai

##### 4.4.5.1. Mô tả

- Khi người dùng yêu cầu dữ liệu, hệ thống sẽ kiểm tra trong Redis cache.
- Nếu có → trả về luôn.
- Nếu không có → đọc từ MongoDB, sau đó lưu vào Redis để các lần sau truy cập nhanh hơn.

##### 4.4.5.2. Code ví dụ

```
1 // server.ts
2 import express from "express";
3 import mongoose from "mongoose";
4 import { createClient } from "redis";
5
6 const app = express();
```

ts

```

7  const port = 3000;
8
9  // Kết nối MongoDB
10 await mongoose.connect("mongodb://localhost:27017/readthrough");
11 const UserSchema = new mongoose.Schema({ name: String, age: Number });
12 const User = mongoose.model("User", UserSchema);
13
14 // Kết nối Redis
15 const redisClient = createClient();
16 redisClient.connect().catch(console.error);
17
18 // Middleware GET user by ID
19 app.get("/users/:id", async (req, res) => {
20   const id = req.params.id;
21
22   // 1. Kiểm tra cache
23   const cachedUser = await redisClient.get(`user:${id}`);
24   if (cachedUser) {
25     console.log("Cache hit");
26     return res.json(JSON.parse(cachedUser));
27   }
28
29   console.log("Cache miss → DB");
30
31   // 2. Nếu không có trong cache → tìm trong DB
32   const user = await User.findById(id);
33   if (!user) return res.status(404).json({ error: "User not found" });
34
35   // 3. Lưu vào Redis cache
36   await redisClient.set(`user:${id}`, JSON.stringify(user), { EX:
37     3600 }); // hết hạn sau 1 giờ
38   return res.json(user);
39 });
40
41 app.listen(port, () => {
42   console.log(`Server running at http://localhost:${port}`);
43 });

```

#### 4.4.5.3. Kiểm thử

- Tạo user trong MongoDB:

```
1 db.users.insertOne({ name: "Alice", age: 25 });
```

js

- Gọi API:

```
1 GET http://localhost:3000/users/<user_id>
```

- Lần đầu sẽ đọc từ DB và lưu vào Redis.
- Lần sau sẽ trả kết quả từ Redis.

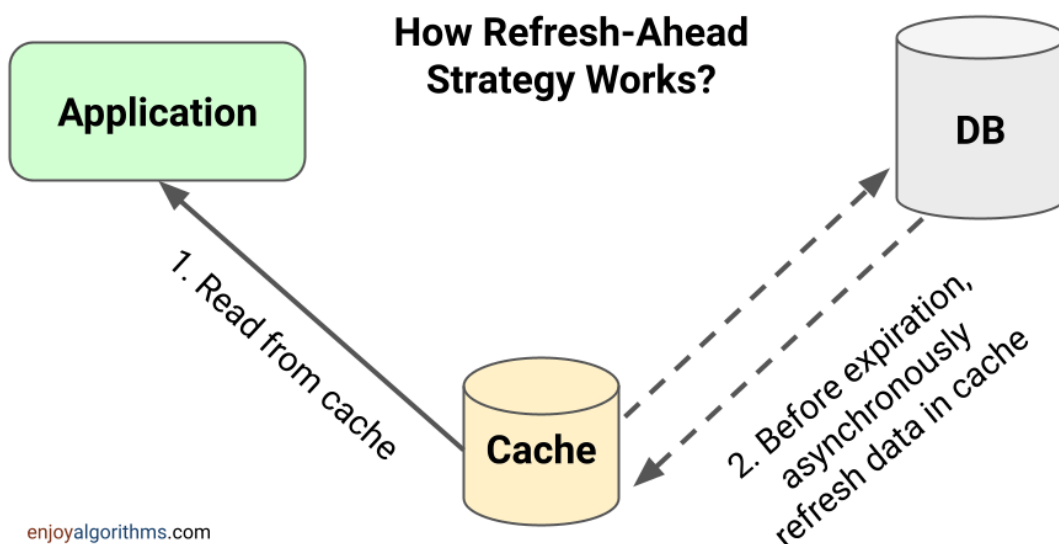
#### 4.4.5.4. Ghi chú

- Đây là **read-through** vì logic truy xuất DB nằm **trong cùng đoạn code xử lý cache**.
- Nếu bạn tách riêng lớp cache, thì gọi là **cache-aside** (ứng dụng tự quyết định khi nào đọc/ghi cache).

## 4.5. Refresh ahead

### 4.5.1. Giới thiệu Refresh Ahead

- Trong các mô hình cache-aside hoặc read-through, nếu dữ liệu trong bộ nhớ đệm bị xóa do hết thời gian sống (TTL - time to live), bộ nhớ đệm sẽ tải lại dữ liệu từ cơ sở dữ liệu khi có yêu cầu đọc tiếp theo cho cùng dữ liệu. Đây là tình huống cache miss, làm tăng độ trễ đọc.
- Câu hỏi đặt ra là: Có cách nào để tải trước dữ liệu (từ cơ sở dữ liệu vào bộ nhớ đệm) trước khi hết TTL, để yêu cầu không cần truy cập cơ sở dữ liệu? Đây là lúc mô hình Refresh Ahead phát huy tác dụng.



#### 4.5.2. Cách hoạt động của Refresh Ahead

- Mục tiêu của mô hình Refresh Ahead là cấu hình bộ nhớ đệm để tự động tải lại (làm mới) phiên bản mới nhất của dữ liệu từ cơ sở dữ liệu một cách bất đồng bộ trước khi hết TTL (hoặc gần đến thời điểm hết TTL). Quá trình làm mới này được thực hiện định kỳ bởi một tác vụ nền hoặc một luồng chuyên dụng.
- Mô hình Refresh Ahead đảm bảo các mục trong bộ nhớ đệm có khả năng được truy cập lại sớm sẽ được chủ động duy trì sẵn sàng trước khi hết hạn. Các yêu cầu đọc dữ liệu tương ứng sẽ được phục vụ từ bộ nhớ đệm với độ trễ thấp.
- Tuy nhiên, có một vấn đề: Nếu làm mới quá thường xuyên, dữ liệu sẽ cập nhật hơn nhưng sẽ tăng tải cho cơ sở dữ liệu. Ngược lại, làm mới không thường xuyên có thể dẫn đến dữ liệu cũ trong bộ nhớ đệm. Vì vậy, cần thiết lập tần suất làm mới phù hợp để cân bằng giữa độ mới của dữ liệu và tải cơ sở dữ liệu.
- Một ý tưởng là xác định **hệ số refresh-ahead** cho các mục trong bộ nhớ đệm. Đây là giá trị từ 0 đến 1, biểu thị phần trăm của TTL. Ví dụ, giả sử TTL của dữ liệu là 120 giây và hệ số refresh-ahead là 0.5:
  - Nếu ứng dụng đọc dữ liệu trước 60 giây, không có gì xảy ra và bộ nhớ đệm trả về dữ liệu.
  - Nếu ứng dụng đọc dữ liệu sau khi hết TTL (sau 120 giây), bộ nhớ đệm sẽ thực hiện đọc đồng bộ từ cơ sở dữ liệu và trả dữ liệu cho ứng dụng. Đọc đồng bộ nghĩa là bộ nhớ đệm sẽ đợi quá trình tải lại hoàn tất trước khi phục vụ dữ liệu, dẫn đến thao tác đọc chậm.
  - Nếu ứng dụng đọc dữ liệu sau 60 giây nhưng trước 120 giây, bộ nhớ đệm trả về giá trị hiện tại và bất đồng bộ lấy phiên bản cập nhật từ cơ sở dữ liệu. Dữ liệu được tải lại trước giới hạn TTL, đồng thời đặt lại TTL (kéo dài thời gian tồn tại trong bộ nhớ đệm). Lưu ý: Quá trình làm mới là bất đồng bộ, nên bộ nhớ đệm không đợi dữ liệu tải lại trước khi phục vụ ứng dụng.

#### 4.5.3. Ưu và nhược điểm của Refresh Ahead

##### 4.5.3.1. Ưu điểm

- Chúng ta thường sử dụng bộ nhớ đệm để lưu trữ dữ liệu được truy cập thường xuyên. Nếu chỉ dựa vào TTL, có nguy cơ phục vụ dữ liệu cũ cho đến khi hết TTL. Ngay cả sau khi hết TTL, việc lấy dữ liệu cập nhật từ cơ sở dữ liệu sẽ tăng độ trễ. Với mô hình Refresh Ahead, các mục được truy cập thường xuyên sẽ được tải lại bất đồng bộ trước khi hết hạn, giúp các lần đọc tiếp theo không gặp độ trễ khi tải từ cơ sở dữ liệu.
- Refresh Ahead đặc biệt hữu ích khi cần cập nhật thường xuyên một lượng lớn dữ liệu được truy cập thường xuyên, và việc giữ dữ liệu mới là yếu tố quan trọng cho hiệu suất và chức năng của ứng dụng.

- Đảm bảo dữ liệu đọc thường xuyên được cập nhật trước khi hết hạn.
- Giảm thời gian chờ khi lấy dữ liệu từ cơ sở dữ liệu, hạ thấp độ trễ đọc cho dữ liệu truy cập thường xuyên so với các kỹ thuật như read-through.
- Giảm các đợt tăng độ trễ đột ngột do hết TTL, giải quyết một phần vấn đề “thundering herd” liên quan đến bộ nhớ đệm read-through.
- Làm mới bất đồng bộ chỉ được kích hoạt khi mục trong bộ nhớ đệm được truy cập và gần hết TTL, tối ưu hóa việc làm mới cho dữ liệu được sử dụng tích cực và tránh tải lại dữ liệu ít truy cập.
- Hữu ích khi dữ liệu được nhiều người dùng truy cập, giữ giá trị mới trong bộ nhớ đệm và tránh độ trễ từ việc tải lại quá nhiều từ cơ sở dữ liệu.

#### 4.5.3.2. Nhược điểm

- Triển khai mô hình này có thể làm tăng độ phức tạp của mã ứng dụng, khó duy trì và gỡ lỗi, đặc biệt nếu logic bộ nhớ đệm trở nên khó quản lý.
- Có thể gây thêm tải cho bộ nhớ đệm và cơ sở dữ liệu nếu nhiều khóa được làm mới cùng lúc.
- Bộ nhớ đệm cần dự đoán chính xác mục nào sẽ cần trong tương lai; dự đoán sai có thể dẫn đến đọc cơ sở dữ liệu không cần thiết.
- Refresh Ahead có thể không phù hợp với mọi mô hình truy cập. Nó hoạt động tốt cho các khối lượng công việc đọc nhiều, nhưng chi phí tải trước có thể không hợp lý cho các mô hình truy cập ghi nhiều.

#### 4.5.3.3. So sánh Refresh Ahead và Read-Through Cache

- **Refresh Ahead:**
  - Refresh Ahead: Tích cực lấy dữ liệu từ lưu trữ trước khi được yêu cầu rõ ràng.
  - Refresh Ahead: Dự đoán và lấy dữ liệu có khả năng được truy cập trong tương lai.
  - Refresh Ahead: Giảm độ trễ truy cập vì dữ liệu đã sẵn sàng trong bộ nhớ đệm khi cần.
  - Refresh Ahead: Có thể gây chi phí nếu hệ thống tiêu tốn nhiều tài nguyên để tải trước dữ liệu, cạnh tranh với các tác vụ khác.
  - Refresh Ahead: Đặc biệt hữu ích cho các mô hình truy cập có thể dự đoán.
- **Read-Through:**
  - Read-Through: Chỉ lấy dữ liệu từ lưu trữ và điền vào bộ nhớ đệm khi ứng dụng yêu cầu rõ ràng.
  - Read-Through: Lấy dữ liệu từ lưu trữ vào bộ nhớ đệm theo yêu cầu khi có lệnh đọc.
  - Read-Through: Ban đầu có thể gặp độ trễ cao hơn.
  - Read-Through: Thường có chi phí thấp hơn vì chỉ lấy dữ liệu khi có yêu cầu đọc thực tế.

- Read-Through: Không có hạn chế như vậy, hoạt động tốt nhất khi mô hình truy cập tương lai ít dự đoán được.

#### 4.5.4. Ví dụ triển khai

##### 4.5.4.1. Mô tả Refresh Ahead

- **TTL (Time-To-Live):** Thời gian sống của 1 cache entry.
- **Refresh time:** Khi đến gần TTL (ví dụ 80% TTL), cache sẽ chủ động **làm mới** dữ liệu.
- **Người dùng vẫn được trả về bản cache cũ** trong lúc dữ liệu được làm mới.

##### 4.5.4.2. TypeScript với giả lập

Giả sử ta có một hàm lấy dữ liệu từ database:

```
1 async function fetchFromDatabase(key: string): Promise<string> {  
2   console.log(`Fetching ${key} from DB...`);  
3   await new Promise((r) => setTimeout(r, 100)); // giả lập độ trễ  
4   return `Data for ${key} at ${new Date().toISOString()}`;  
5 }
```

##### 4.5.4.3. Cache có Refresh Ahead

```
1 type CacheEntry = {  
2   value: string;  
3   expireAt: number;  
4   refreshing: boolean;  
5 };  
6  
7 const cache: Record<string, CacheEntry> = {};  
8 const TTL = 10_000; // 10 giây  
9 const REFRESH_AHEAD_THRESHOLD = 0.8; // 80%  
10  
11 async function getWithRefreshAhead(key: string): Promise<string> {  
12   const now = Date.now();  
13   const entry = cache[key];  
14  
15   if (!entry || now > entry.expireAt) {  
16     // Cache miss hoặc đã hết hạn  
17     const data = await fetchFromDatabase(key);  
18     cache[key] = {  
19       value: data,  
20       expireAt: now + TTL,  
21       refreshing: false,
```

```

22     };
23     return data;
24 }
25
26 // Cache hit
27 const remainingTime = entry.expireAt - now;
28 if (
29     !entry.refreshing &&
30     remainingTime < TTL * (1 - REFRESH_AHEAD_THRESHOLD)
31 ) {
32     // Nếu còn ít hơn 20% thời gian TTL ⇒ bắt đầu làm mới
33     entry.refreshing = true;
34     fetchFromDatabase(key).then((newData) => {
35         cache[key] = {
36             value: newData,
37             expireAt: Date.now() + TTL,
38             refreshing: false,
39         };
40         console.log(`Refreshed ${key} in background.`);
41     });
42 }
43
44 return entry.value;
45 }

```

#### 4.5.4.4. Sử dụng

```

1  async function test() {
2      console.log(await getWithRefreshAhead("user:123")); // Fetch từ DB
3      setInterval(async () => {
4          const data = await getWithRefreshAhead("user:123");
5          console.log("Get:", data);
6      }, 2000); // gọi mỗi 2s
7  }
8
9  test();

```

#### 4.5.4.5. Lợi ích

- Người dùng **luôn nhận được dữ liệu từ cache nhanh chóng**.
- Dữ liệu được **làm mới định kỳ phía sau** => giảm nguy cơ cache miss spike.
- Cân bằng giữa hiệu năng và độ tươi của dữ liệu.

## 4.6. Pre-caching

### 4.6.1. Hướng tiếp cận Pre-Caching

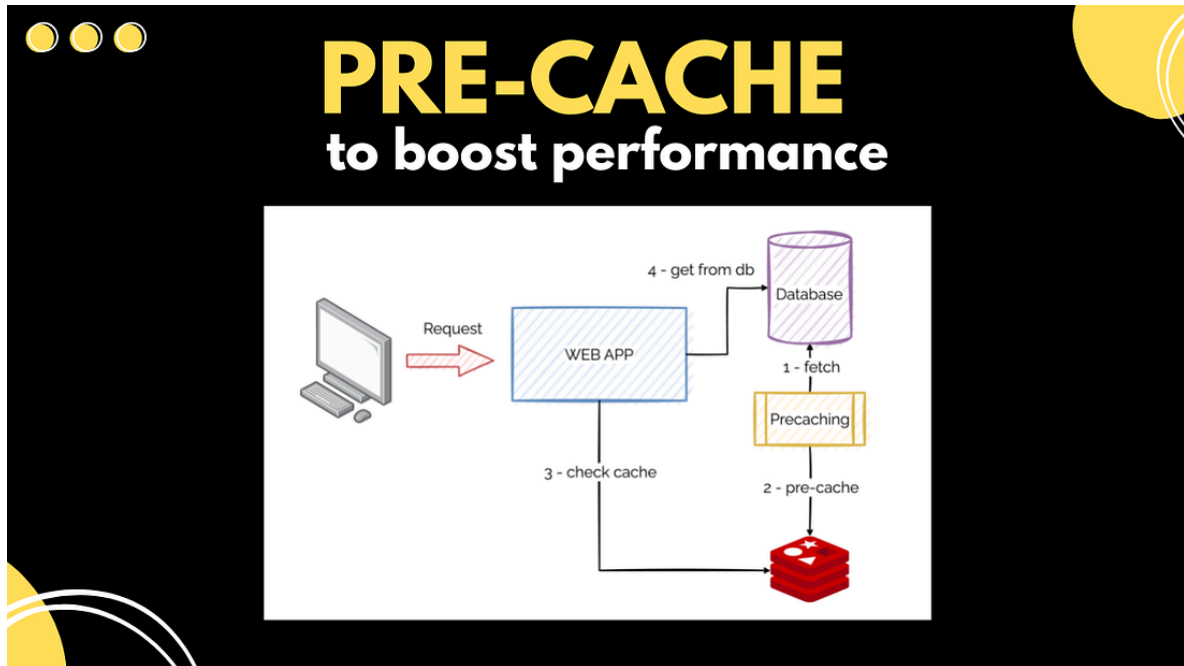
- Tốc độ và hiệu suất là hai yếu tố quan trọng giúp một trang web nổi bật giữa các đối thủ. Hãy tưởng tượng bạn đang truy cập danh sách bán chạy nhất trên trang web Amazon và thấy rằng các trang sản phẩm mất rất lâu để hiển thị.
- Hay một blog đang xuất bản những câu chuyện hay mà độc giả không thể đọc được vì lưu lượng truy cập vượt quá khả năng của máy chủ. Và nếu điều đó chưa đủ, hãy đo lường sự thất vọng của bạn khi có một bộ phim mới được mong đợi trên Netflix và tất cả những gì bạn thấy chỉ là màn hình đang tải.
- Nếu bạn là người đang vận hành một trang web như vậy, đây là một vấn đề tốt để có. Rõ ràng, mọi người đang quan tâm đến những thứ bạn đang cung cấp và có một lượng truy cập không cân đối cho một số trang nhất định.
- Tuy nhiên, nếu không được xử lý đúng cách, tình huống có thể nhanh chóng trở nên hỗn loạn và khiến người dùng của bạn xa lánh. Cuối cùng, nó sẽ dẫn đến mất doanh thu dù là về thời gian xem, doanh số bán hàng hay thậm chí là thiện chí chung của người dùng.
- Một trong những kỹ thuật nổi bật nhất để tăng tốc độ và hiệu suất trang web là **pre-caching** (tiền lưu trữ).

### 4.6.2. Giới thiệu về Pre-caching

- Pre-caching là một kỹ thuật được sử dụng để chủ động lưu trữ hoặc cache dữ liệu dự đoán trước cho các yêu cầu trong tương lai. Ý tưởng là lưu trữ trước dữ liệu hoặc tài nguyên được truy cập phổ biến để khi cần thiết, có thể cung cấp nó cho người dùng cuối nhanh hơn.
- Có thể thực hiện pre-caching ở phía máy khách như trong trình duyệt web. Ngoài ra, cũng có thể thực hiện nó ở phía máy chủ bằng cách sử dụng Mạng Phân phối Nội dung (CDNs) hoặc các giải pháp lưu trữ khác.



- Dù cách tiếp cận là gì, mục tiêu của pre-caching là cải thiện hiệu suất và trải nghiệm người dùng bằng cách giảm thời gian tải nội dung.



#### 4.6.3. Cách hoạt động của Pre-caching

- Quá trình pre-caching đòi hỏi lưu trữ một bản sao của dữ liệu ở vị trí gần người dùng hơn hoặc lưu trữ dữ liệu trước để sẵn sàng khi cần.
  - ▶ Đầu tiên, xác định dữ liệu hoặc tài nguyên được truy cập thường xuyên nhất. Những tài nguyên này là ứng viên tốt cho pre-caching. Ví dụ, các bài đăng blog phổ biến nhất, danh sách sản phẩm bán chạy, v.v. Cũng có thể thêm hình ảnh, tệp JS và bảng định kiểu vào danh sách pre-caching.
  - ▶ Sau khi xác định, cần quyết định hệ thống lưu trữ để lưu trữ dữ liệu đã được pre-cache. Điều này có thể là một bộ nhớ đệm cục bộ trên thiết bị của người dùng hoặc thậm chí là một bộ nhớ đệm phân tán trên nhiều máy chủ. Lựa chọn sẽ phụ thuộc vào loại tài nguyên.
  - ▶ Tiếp theo, cần điền sẵn vào bộ nhớ đệm với các tài nguyên đã xác định. Bước này có thể được hệ thống thực hiện tự động trong giai đoạn khởi tạo. Ngoài ra, có thể thực hiện theo yêu cầu khi dữ liệu được người dùng truy cập. Hãy nhớ rằng, pre-caching là về việc chủ động.
  - ▶ Khi cài đặt đã sẵn sàng, có thể để hệ thống của mình làm việc. Bất cứ khi nào hệ thống cần truy cập dữ liệu đã được pre-cache, nó có thể truy xuất trực tiếp từ bộ nhớ đệm thay vì lấy từ nguồn bên ngoài chậm hơn.

#### 4.6.4. Sử dụng Pre-cache

- Sự thành công của pre-caching phụ thuộc vào chất lượng dữ liệu được pre-cache. Điều quan trọng là phải chọn đúng dữ liệu để lưu trữ.
- Mặc dù việc này có thể nghe có vẻ khó khăn trong thực tế, có thể tuân theo các quy tắc dưới đây để đưa ra lựa chọn đúng đắn:
  - ▶ Ưu tiên các tài nguyên quan trọng như các tệp HTML, CSS và JavaScript cần thiết cho việc tải trang ban đầu. Thông thường, đây là những tài nguyên quan trọng nhất cần thiết để cung cấp trải nghiệm người dùng nhanh chóng và mượt mà.
  - ▶ Cũng nên xem xét việc lưu trữ các tài nguyên của bên thứ ba như phông chữ, thư viện hoặc script từ các miền khác. Những tài nguyên này có thể được pre-cache cục bộ để giảm các yêu cầu mạng thường xuyên.
  - ▶ Giả định ban đầu về các tài nguyên tốt nhất cho pre-caching có thể thay đổi. Do đó, điều quan trọng là phải thực hiện phân tích thường xuyên về mẫu sử dụng ứng dụng web và rút ra thông tin chi tiết về hoạt động của người dùng. Điều này sẽ giúp danh mục pre-caching của luôn phù hợp khi ứng dụng của phát triển.
  - ▶ Khám phá việc sử dụng học máy cho pre-caching. Có thể xây dựng các mô hình dự đoán để dự đoán tài nguyên nào sẽ được yêu cầu trong tương lai dựa trên các mẫu sử dụng trong quá khứ. Có thể đào tạo mô hình này trên dữ liệu lịch sử và sử dụng nó để xác định các tài nguyên ứng cử viên tốt nhất cho pre-caching. Tất nhiên, đây là một cách tiếp cận tốn kém và việc sử dụng nó phụ thuộc vào tầm quan trọng của pre-caching trong bối cảnh ứng dụng.

#### 4.6.5. Ưu và nhược điểm của Pre-caching

##### 4.6.5.1. Ưu điểm

- **Cải thiện hiệu suất** – Khi pre-cache dữ liệu, về cơ bản đang giảm thời gian tải cho nội dung. Điều này dẫn đến trải nghiệm người dùng nhanh hơn. Các trang web và ứng dụng dự kiến có lưu lượng truy cập cao hoặc xử lý lượng dữ liệu lớn được hưởng lợi đáng kể từ điều này.
- **Cải thiện trải nghiệm người dùng** – Ai không thích trải nghiệm người dùng tốt? Thời gian tải nhanh hơn cải thiện trải nghiệm người dùng tổng thể và giúp giảm tỷ lệ thoát (phần trăm người dùng rời khỏi trang web sau khi truy cập một trang). Pre-caching cũng cải thiện khả năng truy cập nội dung ngay cả trong trường hợp kết nối mạng kém.
- **Giảm chi phí** – Pre-caching có thể giúp giảm chi phí. Ví dụ, nếu pre-cache dữ liệu trên CDN, cuối cùng đang giảm tải cho máy chủ gốc. Điều này tiết kiệm băng thông và giảm chi phí máy chủ.

- **Truy cập ngoại tuyến** – Với pre-caching, cũng có thể kích hoạt quyền truy cập ngoại tuyến vào nội dung bằng cách sử dụng khái niệm service workers. Điều này cực kỳ quan trọng đối với các ứng dụng di động và trang web cần hoạt động ở những khu vực có kết nối internet kém.
- **Bảo mật** – Mặc dù đây là một lợi ích gián tiếp, cũng cải thiện bảo mật tổng thể cho tài sản của mình bằng cách sử dụng pre-caching. Về cơ bản, pre-caching làm giảm tác động của các cuộc tấn công DoS (Từ chối Dịch vụ) vì ứng dụng sẽ không phải phục vụ các tài nguyên đã được pre-cache từ máy chủ gốc.

#### 4.6.5.2. Nhược điểm

- **Dữ liệu cũ:** Với pre-caching, đang lưu trữ dữ liệu trước. Dữ liệu này có thể không phải lúc nào cũng cập nhật. Nếu dữ liệu thay đổi thường xuyên, phiên bản đã pre-cache sẽ trở nên lỗi thời và dẫn đến vấn đề. Để tránh tình huống này, phải có chiến lược phù hợp để vô hiệu hóa bộ nhớ đệm nhằm loại bỏ dữ liệu cũ.
- **Sử dụng tài nguyên không hiệu quả:** Trong khi pre-caching, về cơ bản đang giả định rằng dữ liệu đang lưu trữ sẽ được truy cập thường xuyên. Những giả định như vậy có thể không phải lúc nào cũng đúng và có thể kết thúc với dữ liệu không cần thiết thường xuyên. Nếu kích thước của dữ liệu đó vượt quá một giới hạn nhất định, giải pháp lưu trữ có thể trở nên kém hiệu quả và gây lãng phí tài nguyên quý giá có thể được sử dụng cho các mục đích khác.
- **Phạm vi hạn chế:** Pre-caching có phạm vi hạn chế. Nó chỉ áp dụng cho dữ liệu đã biết trước và có thể được điền sẵn trong bộ nhớ đệm. Đây phần lớn là dữ liệu tĩnh. Khó triển khai pre-caching cho dữ liệu được tạo động mà không sử dụng các thuật toán phức tạp.

#### 4.6.6. Phân loại Pre-caching

Vì ý tưởng cơ bản của pre-caching khá đơn giản, có thể triển khai nó theo nhiều cách khác nhau. Nhìn chung, có hai cách tiếp cận chính: pre-caching phía máy khách và pre-caching phía máy chủ.

##### 4.6.6.1. Pre-caching phía máy khách

- Cách phổ biến nhất để pre-caching phía máy khách là chủ động cache tài nguyên trên trình duyệt. Với caching dựa trên trình duyệt, đang cố gắng dự đoán các tài nguyên sẽ được yêu cầu và lưu trữ chúng trước bằng cách sử dụng API cache-storage của trình duyệt.
- Caching dựa trên trình duyệt thường dựa vào các tiêu đề caching để xác định xem một tài nguyên cụ thể có thể được lưu trữ hay không. Khi người dùng yêu cầu một trang, trình duyệt kiểm tra bộ nhớ đệm của nó để xem một bản sao của dữ liệu được

yêu cầu đã có sẵn chưa. Nếu có, trình duyệt tải dữ liệu từ bộ nhớ đệm. Điều này giảm thời gian tải trang.

- Một cách khác để triển khai caching dựa trên trình duyệt là sử dụng API Service Worker. Về cơ bản, tài sản được pre-cache trước, thường là trong quá trình cài đặt service worker.
- Với pre-caching service worker, các tài sản tĩnh và tài liệu quan trọng như tệp HTML, CSS, JS và tệp hình ảnh cần thiết cho truy cập ngoại tuyến được tải xuống và lưu trữ trong một phiên bản Cache. Có thể sử dụng thư viện JavaScript có tên Workbox giúp dễ dàng precache tài nguyên và cung cấp API đơn giản để làm việc với bộ nhớ đệm service worker.

#### 4.6.6.2. Pre-caching phía máy chủ

- Cách tiếp cận thứ hai là pre-caching phía máy chủ. Cũng có thể thực hiện nó bằng nhiều cách.
- Phương pháp đầu tiên là sử dụng Mạng Phân phối Nội dung hoặc CDN lưu trữ một bản sao của dữ liệu trên các máy chủ được phân bố khắp thế giới. Khi người dùng yêu cầu dữ liệu, nó được cung cấp từ máy chủ gần người dùng nhất. Điều này giảm thời gian xử lý yêu cầu và làm cho trang web nhanh hơn đối với người dùng.
- Cách tiếp cận thứ hai là sử dụng Máy chủ Proxy Cache. Đây là một máy chủ nằm phía trước máy chủ gốc và hoạt động như một lớp lưu trữ bằng cách lưu giữ một bản sao của dữ liệu. Lần sau khi có yêu cầu từ người dùng, máy chủ proxy cung cấp dữ liệu trực tiếp mà không cần phải gửi yêu cầu đến máy chủ gốc.

#### 4.6.7. Tối ưu cho Pre-Caching

- Pre-caching là một kỹ thuật cực kỳ hữu ích để cải thiện hiệu suất ứng dụng web. Tuy nhiên, nên tuân theo một số thực hành tốt nhất để đảm bảo rằng pre-caching mang lại kết quả mong muốn.
  - ▶ **Chỉ cache các tài nguyên cần thiết.** Việc lưu trữ quá nhiều tài nguyên có thể dẫn đến lãng phí không gian lưu trữ và làm mất đi lợi ích của pre-caching. Chỉ nên lưu trữ những tài nguyên có tác động lớn nhất đến việc cải thiện hiệu suất.
  - ▶ **Đừng quên phiên bản cho các tài nguyên đã pre-cache.** Ngay cả tài nguyên đã pre-cache cũng có thể được cập nhật trong tương lai. Vì vậy, nên đảm bảo sử dụng phiên bản để xác định phiên bản mới nhất của tài nguyên. Khi một tài nguyên được cập nhật, cũng nên tăng số phiên bản để theo dõi các bản cập nhật tiếp theo.
  - ▶ **Sử dụng các tiêu đề cache-control phù hợp.** Tiêu đề cache-control giúp chúng ta đảm bảo rằng các tài nguyên được lưu trữ trong thời gian hợp lý. Ví dụ, một nền tảng thương mại điện tử có thể có danh sách sản phẩm bán chạy nhất thay đổi thường xuyên do các sản phẩm rơi khỏi bảng xếp hạng hoặc tăng hạng. Những tài

nguyên như vậy nên có thời gian tồn tại trong bộ nhớ đệm ngắn hơn để giữ cho bộ nhớ đệm liên quan.

- ▶ **Sử dụng Mạng Phân phối Nội dung (CDN).** CDN giúp giảm thời gian tải tài nguyên. Nên tận dụng CDN để phân phối tài nguyên đến các máy chủ biên nằm gần người dùng hơn. Máy chủ biên kết hợp với pre-caching là một kỹ thuật mạnh mẽ để tăng cường hiệu suất ứng dụng web.
- ▶ **Sử dụng thư viện hoặc framework để kích hoạt pre-caching.** Mặc dù có thể bị cám dỗ xây dựng giải pháp pre-caching từ đầu, nên xem xét sử dụng thư viện như Workbox để kích hoạt service worker cho việc pre-caching tài nguyên. Đối với caching phía máy chủ, hãy xem xét sử dụng kết hợp CDN và máy chủ proxy cache.

#### 4.6.8. Ví dụ triển khai

##### 4.6.8.1. Tạo Service Worker (service-worker.js)

Trong ví dụ này, chúng ta sẽ tạo một service worker để quản lý cache và pre-cache các tài nguyên khi người dùng truy cập lần đầu.

```
1  // service-worker.js
2
3  const CACHE_NAME = "my-cache-v1";
4  const ASSETS_TO_CACHE = [
5    "/",
6    "/index.html",
7    "/styles.css",
8    "/script.js",
9    "/images/logo.png",
10 ];
11
12 // Pre-caching các tài nguyên khi service worker được cài đặt
13 self.addEventListener("install", (event) => {
14   event.waitUntil(
15     caches.open(CACHE_NAME).then((cache) => {
16       console.log("Pre-caching assets");
17       return cache.addAll(ASSETS_TO_CACHE);
18     })
19   );
20 });
21
22 // Lắng nghe yêu cầu và trả về tài nguyên từ cache hoặc fetch nếu
   không có trong cache
```

```

23 self.addEventListener("fetch", (event) => {
24     event.respondWith(
25         caches.match(event.request).then((cachedResponse) => {
26             // Nếu tìm thấy trong cache, trả về cached response
27             if (cachedResponse) {
28                 return cachedResponse;
29             }
30             // Nếu không, tiến hành fetch tài nguyên từ mạng
31             return fetch(event.request);
32         })
33     );
34 });
35
36 // Xử lý update cache khi service worker cập nhật
37 self.addEventListener("activate", (event) => {
38     const cacheWhitelist = [CACHE_NAME];
39     event.waitUntil(
40         caches.keys().then((cacheNames) => {
41             return Promise.all(
42                 cacheNames.map((cacheName) => {
43                     if (!cacheWhitelist.includes(cacheName)) {
44                         return caches.delete(cacheName);
45                     }
46                 })
47             );
48         })
49     );
50 });

```

#### 4.6.8.2. Đăng ký Service Worker trong File JavaScript Chính

Tiếp theo, bạn cần đăng ký service worker trong file JavaScript chính của ứng dụng, ví dụ main.js:

```

1  // main.js
2
3  if ("serviceWorker" in navigator) {
4      navigator.serviceWorker
5          .register("/service-worker.js")
6          .then((registration) => {
7              console.log("Service Worker registered with scope:",
              registration.scope);

```

```

8    })
9    .catch((error) => {
10      console.log("Service Worker registration failed:", error);
11    });
12  }

```

#### 4.6.8.3. Cập nhật HTML

Đảm bảo rằng bạn đã thêm link tới main.js và các tài nguyên cần thiết trong trang HTML:

```

1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta name="viewport" content="width=device-width, initial-
6        scale=1.0" />
7      <title>Pre-caching Example</title>
8      <link rel="stylesheet" href="/styles.css" />
9    </head>
10   <body>
11     <h1>Welcome to Pre-caching Example</h1>
12     <script src="/main.js"></script>
13   </body>
14 </html>

```

#### 4.6.8.4. Lưu ý

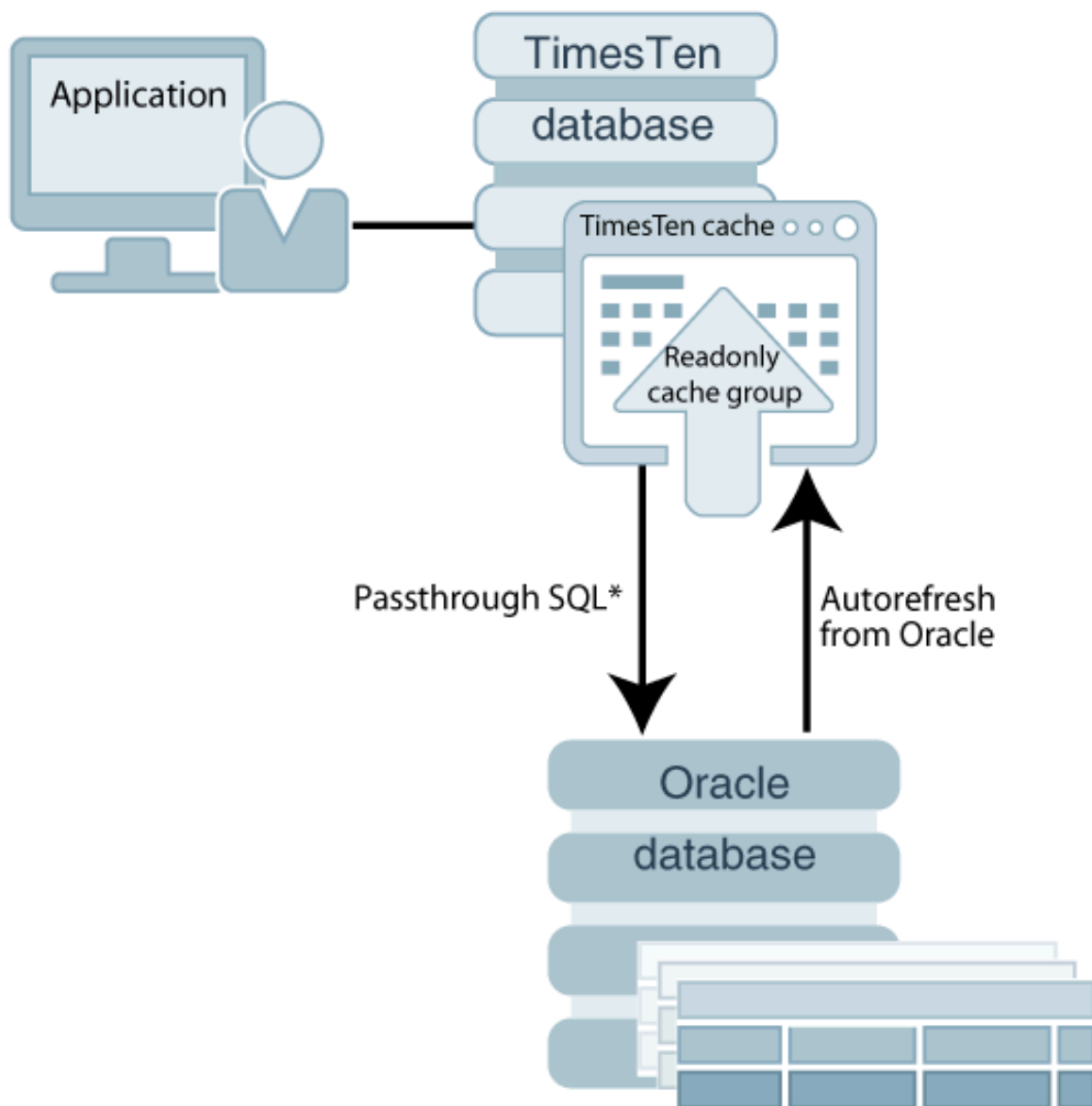
- **Cache Invalidation:** Cần phải lưu ý khi cập nhật tài nguyên, bạn cần phải cập nhật tên cache hoặc sử dụng các chiến lược invalidation cache để tránh việc phục hồi tài nguyên cũ từ cache.
- **Service Worker Lifecycle:** Bạn có thể tìm hiểu thêm về lifecycle của service worker (install, activate, fetch) để có thể quản lý tốt hơn việc pre-caching và cập nhật cache.

### 4.7. Read-only cache

#### 4.7.1. Giới thiệu về Read-Only cache

- **Read-Only Cache** (bộ nhớ đệm chỉ đọc) là một kỹ thuật tối ưu hóa hiệu suất phổ biến, cho phép lưu trữ tạm thời dữ liệu từ nguồn chính ở dạng **chỉ đọc**. Dữ liệu này có thể được truy xuất nhanh chóng, nhưng **không thể bị thay đổi hoặc ghi đè** bởi

ứng dụng hoặc hệ thống đang sử dụng cache.



\* Depending on the PassThrough attribute setting

#### 4.7.2. Đặc điểm chính

Đặc điểm	Mô tả
<b>Chỉ đọc</b>	Dữ liệu cache không thể bị thay đổi sau khi lưu.
<b>Hiệu suất cao</b>	Truy cập nhanh hơn so với nguồn dữ liệu gốc (như cơ sở dữ liệu hoặc ổ đĩa).
<b>Tính ổn định</b>	Giảm nguy cơ sai lệch dữ liệu do thao tác ghi không đồng bộ.
<b>Tự động làm mới</b>	Một số hệ thống có thể tự động cập nhật cache từ nguồn chính.



### 4.7.3. Các ứng dụng và triển khai thực tế

#### 4.7.3.1. IBM – Local Read-Only Cache (LROC)

- **Môi trường áp dụng:** Hệ thống lưu trữ phân tán như IBM Spectrum Scale.
- **Cách hoạt động:** Một thiết bị lưu trữ cục bộ (SSD hoặc HDD) được chỉ định làm cache chỉ đọc.
- **Tính năng:**
  - Cải thiện tốc độ truy cập cho các file thường xuyên đọc.
  - Không cho phép ghi dữ liệu vào thiết bị cache.
  - Tự động đồng bộ dữ liệu từ hệ thống gốc khi cache không còn hợp lệ.

#### 4.7.3.2. Oracle – Read-Only Cache Group (TimesTen)

- **Môi trường áp dụng:** Cơ sở dữ liệu in-memory TimesTen và Oracle DB.
- **Cách hoạt động:**
  - Cache dữ liệu từ các bảng Oracle vào bộ nhớ RAM.
  - Bảng cache thuộc nhóm “read-only” không thể chỉnh sửa.
- **Tính năng:**
  - Dữ liệu được cập nhật định kỳ từ Oracle gốc (refresh).
  - Tăng tốc độ truy vấn đọc trong các ứng dụng xử lý nhanh.

#### 4.7.3.3. Optimizely – Read-Only Object Cache

- **Môi trường áp dụng:** CMS hoặc nền tảng quản lý nội dung.
- **Cách hoạt động:**
  - Sử dụng phương thức `MakeReadOnly()` để chuyển đối tượng thành bất biến.
- **Tính năng:**
  - Ngăn chặn việc sửa đổi các đối tượng trong cache.
  - Hữu ích với các đối tượng dùng chung giữa nhiều user hoặc luồng xử lý.

#### 4.7.3.4. Wikipedia – Page Cache (khái niệm hệ điều hành)

- **Môi trường áp dụng:** Hệ điều hành Linux, Windows, v.v.
- **Cách hoạt động:**
  - Hệ điều hành lưu trữ các “trang” dữ liệu đã đọc từ đĩa trong RAM.
  - Các lần truy xuất sau đó sẽ đọc trực tiếp từ RAM thay vì đĩa.
- **Tính năng:**
  - Tăng tốc độ I/O.
  - Cache này thường không bị sửa đổi trừ khi dữ liệu gốc bị ghi đè.

### 4.7.4. Ưu và nhược điểm của Read-Only Cache

#### 4.7.4.1. Lợi ích của Read-Only Cache

- **Hiệu suất vượt trội:** Giảm độ trễ truy xuất dữ liệu.
- **Tính nhất quán cao:** Không lo xung đột do ghi đồng thời.

- **Tự động cập nhật (nếu có):** Một số hệ thống hỗ trợ tự đồng bộ hóa với nguồn chính.

#### 4.7.4.2. Hạn chế

- **Không ghi được dữ liệu mới:** Không thích hợp cho dữ liệu cần chỉnh sửa thường xuyên.
- **Phụ thuộc vào cơ chế cập nhật:** Nếu không được refresh kịp thời, có thể gây lỗi dữ liệu lỗi thời.
- **Không phù hợp với dữ liệu động:** Ví dụ như giỏ hàng, dữ liệu cá nhân hóa.

#### 4.7.5. Ví dụ triển khai

Để triển khai một **read-only cache** trong TypeScript, bạn có thể tạo một lớp cache mà chỉ cho phép đọc dữ liệu và không cho phép thay đổi hay xóa dữ liệu sau khi đã lưu vào cache. Dưới đây là một ví dụ cơ bản:

```
1  class ReadOnlyCache<K, V> { TS TypeScript
2      private cache: Map<K, V>;
3
4      constructor() {
5          this.cache = new Map<K, V>();
6      }
7
8      // Thêm dữ liệu vào cache (chỉ khi cache chưa có key đó)
9      add(key: K, value: V): void {
10         if (!this.cache.has(key)) {
11             this.cache.set(key, value);
12         }
13     }
14
15     // Lấy giá trị từ cache theo key
16     get(key: K): V | undefined {
17         return this.cache.get(key);
18     }
19
20     // Kiểm tra xem cache có chứa key này hay không
21     has(key: K): boolean {
22         return this.cache.has(key);
23     }
24
25     // Lấy tất cả các keys trong cache
26     keys(): K[] {
27         return Array.from(this.cache.keys());
```

```

28     }
29
30     // Lấy tất cả các values trong cache
31     values(): V[] {
32         return Array.from(this.cache.values());
33     }
34
35     // Lấy tất cả các entry (key-value) trong cache
36     entries(): [K, V][] {
37         return Array.from(this.cache.entries());
38     }
39 }
40
41 // Sử dụng ReadOnlyCache
42 const cache = new ReadOnlyCache<string, number>();
43
44 // Thêm dữ liệu vào cache
45 cache.add("apple", 100);
46 cache.add("banana", 200);
47
48 // Lấy dữ liệu từ cache
49 console.log(cache.get("apple")); // 100
50 console.log(cache.get("banana")); // 200
51
52 // Kiểm tra sự tồn tại của key
53 console.log(cache.has("apple")); // true
54 console.log(cache.has("orange")); // false
55
56 // Lấy tất cả keys và values
57 console.log(cache.keys()); // ["apple", "banana"]
58 console.log(cache.values()); // [100, 200]
59
60 // Không thể xóa hay thay đổi dữ liệu trong cache sau khi đã thêm
61 // cache.add("apple", 150); // Không thể thêm nếu key đã tồn tại
62 // cache.delete("apple"); // Không hỗ trợ xóa trong ReadOnlyCache

```

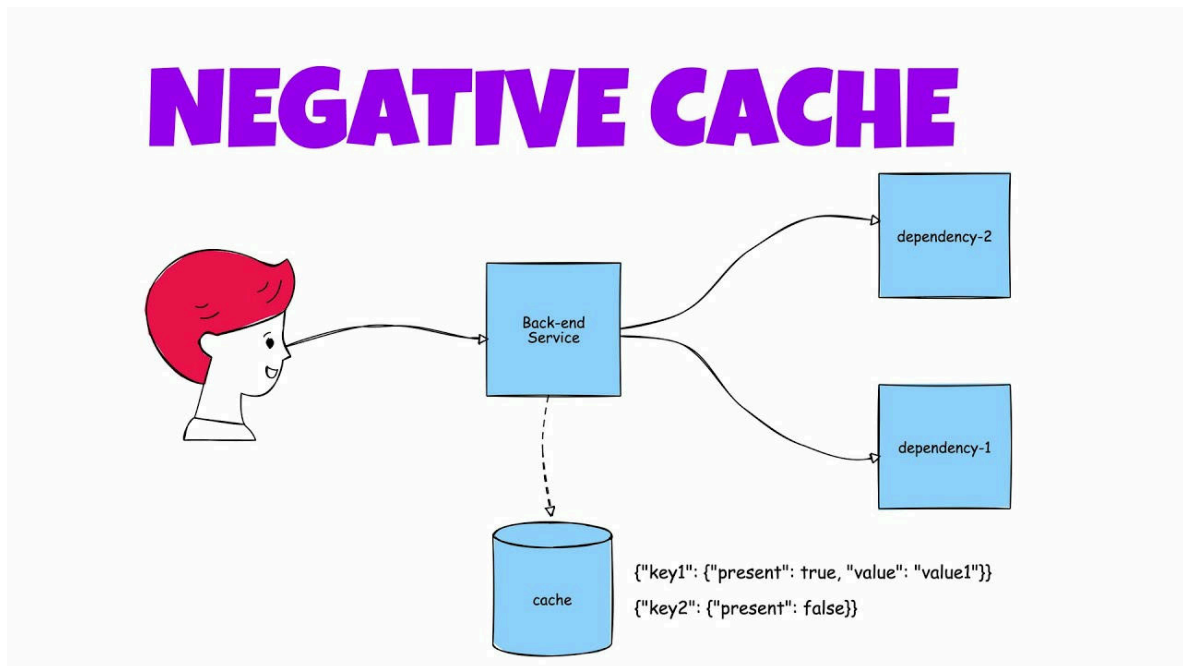
## 4.8. Negative caching

### 4.8.1. Giới thiệu về Negative Caching

- Negative caching là một kỹ thuật caching nâng cao, trong đó hệ thống không chỉ lưu trữ các phản hồi thành công mà còn lưu trữ thông tin về các yêu cầu thất bại hoặc

không tồn tại. Trong khi caching truyền thống tập trung vào việc lưu trữ dữ liệu “positive” (dữ liệu tồn tại và có thể truy xuất thành công), negative caching mở rộng khái niệm này bằng cách thêm vào việc lưu trữ các phản hồi “negative” (dữ liệu không tồn tại hoặc không thể truy xuất).

- Negative caching là quá trình lưu trữ tạm thời kết quả của các truy vấn thất bại hoặc trả về kết quả trống trong một khoảng thời gian nhất định, nhằm tránh việc phải xử lý lại các yêu cầu tương tự trong tương lai.



#### 4.8.2. Nguyên lý hoạt động

- Khi một hệ thống nhận được yêu cầu và xác định rằng tài nguyên được yêu cầu không tồn tại hoặc không khả dụng, thay vì chỉ trả về phản hồi lỗi và quên đi, hệ thống sẽ:
  - Lưu trữ thông tin về yêu cầu thất bại này vào cache
  - Đặt thời gian sống (TTL - Time To Live) cho bản ghi negative cache này
  - Sử dụng bản ghi đã lưu để phản hồi ngay lập tức cho các yêu cầu tương tự tiếp theo

#### 4.8.3. Ưu và nhược điểm của Negative Caching

##### 4.8.3.1. Ưu điểm

- Cải thiện hiệu suất hệ thống
  - **Giảm độ trễ:** Trả về kết quả lỗi đã được lưu trong cache nhanh chóng, không cần phải xử lý lại yêu cầu.
  - **Giảm tải cho hệ thống backend:** Đặc biệt quan trọng khi có nhiều yêu cầu đến tài nguyên không tồn tại.

- ▶ **Tối ưu hóa băng thông:** Giảm lượng truy vấn không cần thiết đến các nguồn dữ liệu.
- Nâng cao bảo mật và ổn định
  - ▶ **Bảo vệ khỏi tấn công DoS:** Ngăn chặn các cuộc tấn công cố ý gửi nhiều yêu cầu đến tài nguyên không tồn tại nhằm làm quá tải hệ thống.
  - ▶ **Tăng khả năng chịu lỗi:** Giảm thiểu tác động khi hệ thống backend gặp sự cố, vẫn có thể trả về thông tin từ cache.
- Tối ưu hóa chi phí
  - ▶ **Giảm chi phí xử lý:** Đặc biệt quan trọng trong môi trường điện toán đám mây, nơi chi phí được tính theo tài nguyên sử dụng.
  - ▶ **Giảm chi phí truy vấn cơ sở dữ liệu:** Nhiều hệ thống cơ sở dữ liệu tính phí theo số lượng truy vấn.

#### 4.8.4. Ứng dụng của Negative Caching

##### 4.8.4.1. DNS Negative Caching

- DNS (Domain Name System) là một trong những ứng dụng phổ biến nhất của negative caching. Khi một máy chủ DNS nhận được truy vấn về một tên miền không tồn tại, nó sẽ lưu trữ kết quả “không tồn tại” này trong cache:
  - ▶ **Định nghĩa NXDOMAIN:** Khi một tên miền không tồn tại, máy chủ DNS sẽ trả về phản hồi NXDOMAIN (Non-Existent Domain).
  - ▶ **RFC 2308:** Tiêu chuẩn này định nghĩa cách thức negative caching trong DNS.
  - ▶ **Giảm tải cho root servers:** Đặc biệt quan trọng cho hệ thống DNS toàn cầu.

##### 4.8.4.2. CDN (Content Delivery Network)

- Các nhà cung cấp CDN như Cloudflare, Akamai và Fastly đều triển khai negative caching:
  - ▶ **Cache lỗi HTTP:** Lưu trữ các mã phản hồi lỗi như 404 (Not Found), 410 (Gone), 500 (Internal Server Error).
  - ▶ **Giảm tải cho origin server:** Ngăn chặn các yêu cầu không cần thiết đến máy chủ gốc.
  - ▶ **Cấu hình tùy chỉnh:** Cho phép quản trị viên cấu hình thời gian sống cho các loại phản hồi lỗi khác nhau.

##### 4.8.4.3. Bộ nhớ đệm ứng dụng Web

- Trong các ứng dụng web hiện đại, negative caching được sử dụng ở nhiều cấp độ:
  - ▶ **Cache API:** Lưu trữ các phản hồi API không thành công.
  - ▶ **Cache cơ sở dữ liệu:** Lưu trữ kết quả truy vấn trả về không có dữ liệu.

- ▶ **Cache dữ liệu người dùng:** Lưu trữ thông tin về tài nguyên người dùng không có quyền truy cập.

#### 4.8.4.4. Proxy Servers và Load Balancers

- Các proxy server và load balancer thường triển khai negative caching để:
  - ▶ **Giảm tải backend:** Ngăn chặn các yêu cầu không cần thiết đến các máy chủ backend.
  - ▶ **Xử lý lỗi hiệu quả:** Trả về phản hồi lỗi nhanh chóng cho các yêu cầu tương tự.
  - ▶ **Quản lý tài nguyên tốt hơn:** Phân bổ tài nguyên hiệu quả hơn cho các yêu cầu hợp lệ.

### 4.8.5. Triển khai Negative Caching

#### 4.8.5.1. Các tham số cấu hình quan trọng

- Khi triển khai negative caching, cần xem xét các tham số cấu hình sau:
  - ▶ **TTL (Time To Live):** Thời gian mà một bản ghi negative cache có hiệu lực. Thường ngắn hơn TTL của positive cache.
  - ▶ **Kích thước cache:** Số lượng bản ghi negative cache tối đa có thể lưu trữ.
  - ▶ **Chính sách loại bỏ:** Thuật toán để quyết định bản ghi nào bị loại bỏ khi cache đầy (ví dụ: LRU - Least Recently Used).
  - ▶ **Loại phản hồi lỗi:** Xác định loại lỗi nào cần được lưu vào cache (ví dụ: chỉ lưu 404, không lưu 500).

#### 4.8.5.2. Ví dụ triển khai

##### 4.8.5.2.1. Cấu hình Negative Caching trong NGINX

```
1 proxy_cache_path /path/to/cache levels=1:2
   keys_zone=my_cache:10m inactive=60m;
2 proxy_cache_valid 200 302 10m;
3 proxy_cache_valid 404 1m;           # Lưu cache lỗi 404 trong 1 phút
```

nginx Nginx

##### 4.8.5.2.2. Cấu hình Negative Caching trong Varnish

```
1 sub vcl_backend_response {
2     if (beresp.status == 404) {
3         set beresp.ttl = 30s; # Lưu cache lỗi 404 trong 30 giây
4         set beresp.uncacheable = false;
5         return(deliver);
6     }
7 }
```

vcl


#### 4.8.5.2.3. Cấu hình Negative Caching trong DNS (BIND)

```
1 options {
2     directory "/var/named";
3     max-ncache-ttl 3600; # Thời gian tối đa lưu trữ negative cache (1
    giờ)
4 };
```

#### 4.8.5.3. Cài đặt trong các ngôn ngữ lập trình


##### 4.8.5.3.1. Java với Caffeine Cache

```
1 LoadingCache<String, Optional<Data>> cache =
  Caffeine.newBuilder()
2     .expireAfterWrite(10, TimeUnit.MINUTES)
3     .build(key -> {
4         Data data = dataService.fetchData(key);
5         return Optional.ofNullable(data); // Lưu Optional.empty() cho
        dữ liệu không tồn tại
6     });
7
8 // Sử dụng cache
9 Optional<Data> result = cache.get(key);
10 if (result.isPresent()) {
11     // Xử lý dữ liệu tồn tại
12 } else {
13     // Xử lý dữ liệu không tồn tại (từ negative cache)
14 }
```

 Java

##### 4.8.5.3.2. Python với Redis

```
1 import redis
2 import json
3
4 r = redis.Redis(host='localhost', port=6379, db=0)
5
6 def get_data(key):
7     # Kiểm tra trong cache
8     cached_value = r.get(key)
9
10     if cached_value is not None:
11         if cached_value == b"__NEGATIVE__":
12             # Negative cache hit
13             return None
```

 Python

```

14         else:
15             # Positive cache hit
16             return json.loads(cached_value)
17
18         # Cache miss, truy vấn từ nguồn
19         data = fetch_from_source(key)
20
21         if data is None:
22             # Lưu negative cache với TTL ngắn hơn
23             r.setex(key, 60, "__NEGATIVE__") # TTL 60 giây
24         else:
25             # Lưu positive cache với TTL dài hơn
26             r.setex(key, 300, json.dumps(data)) # TTL 300 giây
27
28         return data

```

## 4.8.6. Thách thức và Cân nhắc khi triển khai

### 4.8.6.1. Quản lý TTL

- Một trong những thách thức lớn nhất khi triển khai negative caching là xác định thời gian sống phù hợp:
  - **TTL quá ngắn:** Không mang lại lợi ích tối đa của negative caching.
  - **TTL quá dài:** Có thể gây ra tình trạng stale data khi tài nguyên đã tồn tại nhưng vẫn bị coi là không tồn tại.
- Khuyến nghị: Sử dụng TTL ngắn hơn cho negative cache so với positive cache. Thông thường, TTL cho negative cache trong khoảng từ vài giây đến vài phút.

### 4.8.6.2. Xử lý sự cố

- Cần có chiến lược xử lý sự cố khi triển khai negative caching:
  - **Khả năng xóa cache:** Cần có cơ chế để xóa negative cache khi cần thiết.
  - **Monitoring:** Theo dõi tỷ lệ negative cache hits để phát hiện vấn đề.
  - **Circuit breaker:** Tạm thời vô hiệu hóa negative caching trong trường hợp phát hiện bất thường.

### 4.8.6.3. Tính nhất quán dữ liệu

- Negative caching có thể gây ra vấn đề về tính nhất quán dữ liệu:
  - **Thay đổi trạng thái:** Khi tài nguyên chuyển từ không tồn tại sang tồn tại.
  - **Phân phối hệ thống:** Trong hệ thống phân tán, negative cache có thể không đồng bộ giữa các nút.
- Giải pháp: Sử dụng cơ chế invalidation cache hoặc TTL ngắn cho negative cache.



#### 4.8.7. Best Practices và Khuyến nghị

##### 4.8.7.1. Sử dụng TTL phù hợp

- **DNS:** 5-30 phút cho phản hồi NXDOMAIN, tùy thuộc vào môi trường.
- **HTTP:** 1-5 phút cho lỗi 404, 10-30 giây cho lỗi 500.
- **Ứng dụng:** 30-60 giây cho truy vấn dữ liệu không tồn tại.

##### 4.8.7.2. Phân biệt các loại lỗi

- Không phải tất cả các lỗi đều nên được lưu vào negative cache:
  - **Lỗi không tồn tại (404):** Phù hợp cho negative caching.
  - **Lỗi hệ thống (500):** Cần cân trọng, nên sử dụng TTL ngắn hoặc không cache.
  - **Lỗi xác thực (401, 403):** Có thể cache nhưng cần xem xét ngữ cảnh sử dụng.

##### 4.8.7.3. Tối ưu hóa kích thước cache

- **Giới hạn kích thước:** Đặt giới hạn hợp lý cho kích thước negative cache.
- **Sử dụng key hợp lý:** Thiết kế key cache sao cho có thể phân biệt giữa các loại phản hồi lỗi khác nhau.
- **Nén dữ liệu:** Giảm kích thước của các bản ghi negative cache.

##### 4.8.7.4. Kết hợp với các kỹ thuật khác

- Negative caching hoạt động tốt nhất khi kết hợp với các kỹ thuật khác:
  - **Rate limiting:** Giới hạn số lượng yêu cầu đến các tài nguyên không tồn tại.
  - **Circuit breaker:** Ngăn chặn cascade failure khi hệ thống backend gặp sự cố.
  - **Graceful degradation:** Cung cấp trải nghiệm người dùng tốt nhất có thể ngay cả khi có lỗi.

#### 4.8.8. Xu hướng và phát triển

##### 4.8.8.1. Edge Computing và Negative Caching

- Negative caching đang trở nên quan trọng trong kiến trúc edge computing:
  - **Decentralized caching:** Negative cache được phân tán đến các edge server.
  - **Locality awareness:** Negative cache được tối ưu hóa dựa trên vị trí địa lý.
  - **Real-time invalidation:** Hệ thống invalidation cache thời gian thực.

##### 4.8.8.2. Machine Learning trong Negative Caching

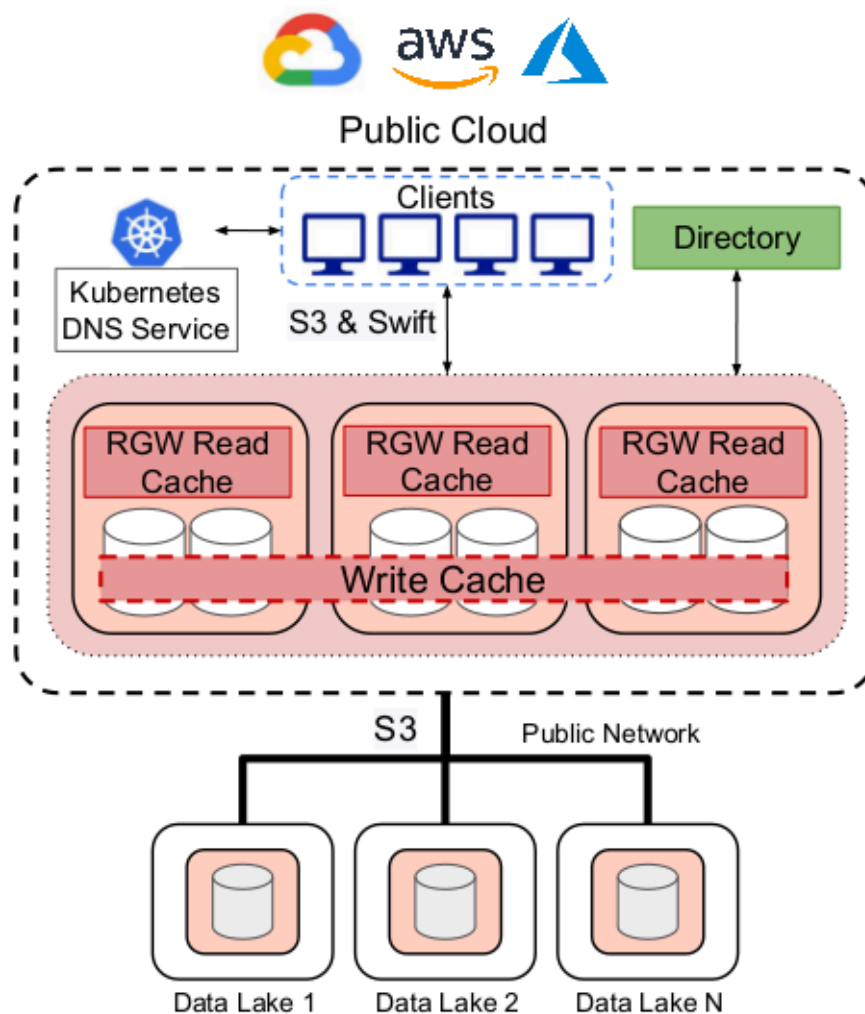
- Các hệ thống hiện đại đang áp dụng machine learning để tối ưu hóa negative caching:
  - **Dự đoán TTL:** Sử dụng ML để dự đoán thời gian sống tối ưu cho các bản ghi negative cache.
  - **Phát hiện bất thường:** Sử dụng ML để phát hiện mẫu yêu cầu bất thường và điều chỉnh chính sách negative caching.

- **Tối ưu hóa tự động:** Tự động điều chỉnh tham số negative caching dựa trên mẫu truy cập.

## 4.9. Hybrid caching

### 4.9.1. Giới thiệu về Hybrid Caching

- Hybrid caching là một phương pháp caching tiên tiến kết hợp hai hoặc nhiều kỹ thuật caching khác nhau nhằm tối ưu hóa hiệu suất, độ tin cậy và chi phí của hệ thống. Trong thời đại số hóa ngày nay, khi khối lượng dữ liệu và số lượng người dùng tăng đột biến, các hệ thống đơn lẻ thường không đáp ứng được đầy đủ các yêu cầu về hiệu suất và khả năng mở rộng. Hybrid caching ra đời như một giải pháp toàn diện, tận dụng ưu điểm của các phương pháp caching khác nhau đồng thời giảm thiểu nhược điểm của chúng.
- Hybrid caching là chiến lược lưu trữ tạm thời dữ liệu bằng cách kết hợp nhiều loại bộ nhớ cache hoặc nhiều phương pháp caching khác nhau để đạt được hiệu suất tối ưu, cân bằng giữa tốc độ truy xuất, dung lượng lưu trữ, chi phí và độ tin cậy của hệ



thống.

#### 4.9.1.1. Bản chất của Hybrid Caching

- Cốt lõi của hybrid caching là tận dụng các đặc điểm khác biệt của các loại bộ nhớ và phương pháp caching:
  - **Đa tầng về phần cứng:** Kết hợp các loại bộ nhớ vật lý khác nhau như RAM, SSD, HDD, mạng.
  - **Đa dạng về thuật toán:** Kết hợp các thuật toán caching như LRU, LFU, ARC, FIFO.
  - **Đa chiến lược về phân phối:** Kết hợp local caching, distributed caching, global caching.
  - **Đa mô hình về dữ liệu:** Kết hợp in-memory caching, persistent caching, database caching.

#### 4.9.1.2. Sự cần thiết của Hybrid Caching

- Trong thực tế, các ứng dụng hiện đại phải đối mặt với nhiều thách thức:
  - **Đa dạng về mẫu truy cập dữ liệu:** Một số dữ liệu được truy cập thường xuyên, một số khác rất ít khi được truy cập.
  - **Yêu cầu về độ trễ thấp:** Người dùng mong đợi thời gian phản hồi gần như tức thì.
  - **Khối lượng dữ liệu lớn:** Không thể lưu trữ tất cả dữ liệu trong bộ nhớ đắt tiền như RAM.
  - **Phân tán địa lý:** Người dùng truy cập từ nhiều vị trí khác nhau trên toàn cầu.
- Hybrid caching giải quyết những thách thức này bằng cách cung cấp một giải pháp toàn diện, linh hoạt và có thể mở rộng.

### 4.9.2. Các kiểu Hybrid Caching phổ biến

#### 4.9.2.1. Multi-level Cache (Cache đa tầng)

- Multi-level cache là một trong những dạng hybrid caching phổ biến nhất, tổ chức cache theo các tầng với hiệu suất và dung lượng khác nhau.

##### 4.9.2.1.1. Nguyên lý hoạt động

- **L1 Cache (Tầng 1):** Bộ nhớ nhanh nhất, dung lượng nhỏ nhất, thường là RAM.
- **L2 Cache (Tầng 2):** Dung lượng lớn hơn, tốc độ chậm hơn, có thể là SSD.
- **L3 Cache (Tầng 3):** Dung lượng lớn nhất, tốc độ chậm nhất, có thể là HDD hoặc mạng.
- Khi một yêu cầu dữ liệu đến, hệ thống kiểm tra lần lượt từ L1 đến Ln. Nếu không tìm thấy ở tầng nào, hệ thống sẽ lấy dữ liệu từ nguồn gốc (database, API) và lưu vào các tầng cache.

#### 4.9.2.1.2. Ưu điểm

- Cân bằng giữa hiệu suất và chi phí.
- Tối ưu hóa việc sử dụng tài nguyên.
- Cải thiện hit rate tổng thể của hệ thống.

#### 4.9.2.1.3. Ví dụ triển khai

```
1 // Minh họa Multi-level Cache trong Java
2 public class MultiLevelCache<K, V> {
3     private Cache<K, V> l1Cache; // In-memory cache (nhanh)
4     private Cache<K, V> l2Cache; // Disk-based cache (chậm hơn)
5
6     public V get(K key) {
7         // Kiểm tra L1 cache
8         V value = l1Cache.get(key);
9         if (value != null) {
10             return value; // L1 cache hit
11         }
12
13         // Kiểm tra L2 cache
14         value = l2Cache.get(key);
15         if (value != null) {
16             // Thêm vào L1 cache cho lần sau
17             l1Cache.put(key, value);
18             return value; // L2 cache hit
19         }
20
21         return null; // Cache miss
22     }
23
24     public void put(K key, V value) {
25         // Lưu vào cả hai tầng cache
26         l1Cache.put(key, value);
27         l2Cache.put(key, value);
28     }
29 }
```

#### 4.9.2.2. Distributed Hybrid Cache (Cache phân tán hỗn hợp)

- Distributed hybrid cache kết hợp local caching và distributed caching để tận dụng ưu điểm của cả hai phương pháp.

#### 4.9.2.2.1. Nguyên lý hoạt động

- **Local Cache:** Mỗi nút trong hệ thống duy trì một bộ nhớ cache cục bộ (thường là in-memory).
- **Distributed Cache:** Cache được phân tán trên nhiều nút (sử dụng Redis, Memcached, Hazelcast).
- **Quy trình truy xuất:** Kiểm tra local cache trước, nếu cache miss thì kiểm tra distributed cache, nếu vẫn miss thì lấy từ nguồn dữ liệu gốc.

#### 4.9.2.2.2. Ưu điểm

- Giảm độ trễ khi truy xuất dữ liệu phổ biến (local cache).
- Cung cấp khả năng mở rộng và chia sẻ dữ liệu (distributed cache).
- Giảm tải cho hệ thống distributed cache.

#### 4.9.2.2.3. Ví dụ triển khai

```
1 // Minh họa Distributed Hybrid Cache trong Java
2 public class DistributedHybridCache<K, V> {
3     private Cache<K, V> localCache; // Cache cục bộ
4     private DistributedCache<K, V> distributedCache; // Cache phân tán
5     private DataSource<K, V> dataSource; // Nguồn dữ liệu gốc
6
7     public V get(K key) {
8         // Bước 1: Kiểm tra local cache
9         V value = localCache.get(key);
10        if (value != null) {
11            return value; // Local cache hit
12        }
13
14        // Bước 2: Kiểm tra distributed cache
15        value = distributedCache.get(key);
16        if (value != null) {
17            // Cập nhật local cache
18            localCache.put(key, value);
19            return value; // Distributed cache hit
20        }
21
22        // Bước 3: Lấy từ nguồn dữ liệu gốc
23        value = dataSource.get(key);
24        if (value != null) {
25            // Cập nhật cả hai loại cache
26            localCache.put(key, value);
```

```

27         distributedCache.put(key, value);
28     }
29
30     return value;
31 }
32 }

```

#### 4.9.2.3. Write-Through và Write-Back Hybrid Cache

- Đây là hybrid caching kết hợp các chiến lược ghi khác nhau để cân bằng giữa độ tin cậy và hiệu suất.

##### 4.9.2.3.1. Nguyên lý hoạt động

- **Write-Through:** Dữ liệu được ghi đồng thời vào cache và nguồn dữ liệu gốc.
- **Write-Back (Write-Behind):** Dữ liệu được ghi vào cache trước, sau đó định kỳ hoặc khi được kích hoạt mới ghi vào nguồn dữ liệu gốc.
- **Kết hợp:** Một số dữ liệu quan trọng sử dụng write-through để đảm bảo độ tin cậy, trong khi dữ liệu ít quan trọng sử dụng write-back để tối ưu hiệu suất.

##### 4.9.2.3.2. Ưu điểm

- Linh hoạt trong việc xử lý dữ liệu với các mức độ quan trọng khác nhau.
- Cân bằng giữa hiệu suất và độ tin cậy.
- Giảm tải cho hệ thống lưu trữ chính trong thời điểm cao điểm.

##### 4.9.2.3.3. Ví dụ triển khai

```

1  // Minh họa Write-Through và Write-Back Hybrid Cache
2  public class HybridWriteCache<K, V> {
3      private Cache<K, V> cache;
4      private DataStore<K, V> dataStore;
5      private Map<K, V> writeBackBuffer = new ConcurrentHashMap<>();
6      private boolean isHighPriority(K key) {
7          // Logic để xác định dữ liệu có độ ưu tiên cao hay không
8          return key.toString().startsWith("CRITICAL_");
9      }
10
11     public void put(K key, V value) {
12         // Lưu vào cache trước
13         cache.put(key, value);
14
15         if (isHighPriority(key)) {
16             // Write-Through cho dữ liệu quan trọng
17             dataStore.put(key, value);

```

```

18         } else {
19             // Write-Back cho dữ liệu thông thường
20             writeBackBuffer.put(key, value);
21         }
22     }
23
24     // Gọi định kỳ hoặc khi shutdown
25     public void flushWriteBackBuffer() {
26         for (Map.Entry<K, V> entry : writeBackBuffer.entrySet()) {
27             dataStore.put(entry.getKey(), entry.getValue());
28         }
29         writeBackBuffer.clear();
30     }
31 }

```

#### 4.9.2.4. Multi-Algorithm Cache (Cache đa thuật toán)

- Multi-Algorithm Cache sử dụng nhiều thuật toán thay thế (replacement algorithms) khác nhau cho các loại dữ liệu khác nhau.

##### 4.9.2.4.1. Nguyên lý hoạt động

- **Phân loại dữ liệu:** Dữ liệu được phân loại dựa trên đặc điểm truy cập.
- **Áp dụng thuật toán phù hợp:** Mỗi loại dữ liệu sử dụng thuật toán cache thay thế tối ưu.
  - LRU (Least Recently Used): Cho dữ liệu có tính thời gian cao.
  - LFU (Least Frequently Used): Cho dữ liệu có tần suất truy cập ổn định.
  - FIFO (First In First Out): Cho dữ liệu có chu kỳ cập nhật đều đặn.
  - ARC (Adaptive Replacement Cache): Tự động điều chỉnh giữa recency và frequency.

##### 4.9.2.4.2. Ưu điểm


- Tối ưu hóa hit rate cho từng loại dữ liệu.
- Thích ứng với các mẫu truy cập khác nhau.
- Sử dụng hiệu quả không gian cache.

##### 4.9.2.4.3. Ví dụ triển khai

```

1  # Minh họa Multi-Algorithm Cache trong Python
2  class MultiAlgorithmCache:
3      def __init__(self):
4          self.lru_cache = LRUCache(1000) # Cho dữ liệu temporal
5          self.lfu_cache = LFUCache(1000) # Cho dữ liệu frequency-based
6          self.arc_cache = ARCCache(1000) # Cho dữ liệu hỗn hợp

```

 Python

```

7
8     def get_cache_type(self, key):
9         # Logic để xác định loại cache phù hợp
10        if key.startswith("temp_"):
11            return "lru"
12        elif key.startswith("freq_"):
13            return "lfu"
14        else:
15            return "arc"
16
17    def get(self, key):
18        cache_type = self.get_cache_type(key)
19
20        if cache_type == "lru":
21            return self.lru_cache.get(key)
22        elif cache_type == "lfu":
23            return self.lfu_cache.get(key)
24        else:
25            return self.arc_cache.get(key)
26
27    def put(self, key, value):
28        cache_type = self.get_cache_type(key)
29
30        if cache_type == "lru":
31            self.lru_cache.put(key, value)
32        elif cache_type == "lfu":
33            self.lfu_cache.put(key, value)
34        else:
35            self.arc_cache.put(key, value)

```

#### 4.9.2.5. Cache Tiering kết hợp In-Memory và Persistent Cache

- Kết hợp in-memory cache (RAM) và persistent cache (disk) để cân bằng giữa tốc độ và khả năng lưu trữ dài hạn.

##### 4.9.2.5.1. Nguyên lý hoạt động

- **In-Memory Tier:** Lưu trữ dữ liệu truy cập thường xuyên trong RAM.
- **Persistent Tier:** Lưu trữ dữ liệu ít truy cập hơn hoặc dữ liệu lớn trên đĩa.
- **Cache Warming:** Khi khởi động, nạp dữ liệu từ persistent cache vào in-memory cache.
- **Cache Persistence:** Định kỳ lưu snapshot của in-memory cache xuống persistent cache.



#### 4.9.2.5.2. Ưu điểm

- Khả năng phục hồi sau khi hệ thống khởi động lại.
- Cân bằng giữa hiệu suất và dung lượng lưu trữ.
- Tiết kiệm RAM cho dữ liệu ít được truy cập.

#### 4.9.2.5.3. Ví dụ triển khai

```
1 // Minh họa Cache Tiering trong Java
2 public class TieredCache<K, V> {
3     private InMemoryCache<K, V> memoryCache;
4     private DiskCache<K, V> diskCache;
5
6     public TieredCache() {
7         this.memoryCache = new InMemoryCache<>(1000); // Giới hạn 1000
            entries
8         this.diskCache = new DiskCache<>("cache-data");
9
10        // Cache warming khi khởi động
11        loadFromDisk();
12    }
13
14    public V get(K key) {
15        // Kiểm tra in-memory cache
16        V value = memoryCache.get(key);
17        if (value != null) {
18            return value;
19        }
20
21        // Kiểm tra disk cache
22        value = diskCache.get(key);
23        if (value != null) {
24            // Thêm vào memory cache cho truy cập sau
25            memoryCache.put(key, value);
26        }
27
28        return value;
29    }
30
31    public void put(K key, V value) {
32        memoryCache.put(key, value);
33        diskCache.put(key, value);
34    }
```

```

35
36     // Lưu snapshot của memory cache xuống disk
37     public void saveSnapshot() {
38         Map<K, V> snapshot = memoryCache.getAll();
39         for (Map.Entry<K, V> entry : snapshot.entrySet()) {
40             diskCache.put(entry.getKey(), entry.getValue());
41         }
42     }
43
44     // Nạp dữ liệu từ disk vào memory khi khởi động
45     private void loadFromDisk() {
46         Map<K, V> data = diskCache.getAll();
47         for (Map.Entry<K, V> entry : data.entrySet()) {
48             memoryCache.put(entry.getKey(), entry.getValue());
49         }
50     }
51 }

```

### 4.9.3. Các thành phần của một hệ thống Hybrid Caching

#### 4.9.3.1. Cache Storage (Nơi lưu trữ cache)

##### 4.9.3.1.1. In-Memory Cache

- **RAM-based:** Sử dụng bộ nhớ RAM để lưu trữ dữ liệu.
- **Ưu điểm:** Tốc độ truy xuất cực nhanh (nano-seconds).
- **Nhược điểm:** Dung lượng hạn chế, mất dữ liệu khi mất điện hoặc restart.
- **Công nghệ phổ biến:** Memcached, Redis (có thể cấu hình chỉ in-memory), Caffeine (Java), caches tích hợp trong frameworks.

##### 4.9.3.1.2. Persistent Cache

- **Disk-based:** Sử dụng SSD hoặc HDD để lưu trữ dữ liệu.
- **Ưu điểm:** Dung lượng lớn, dữ liệu không bị mất khi restart.
- **Nhược điểm:** Tốc độ chậm hơn RAM (micro-seconds đến milli-seconds).
- **Công nghệ phổ biến:** Redis (persistent mode), RocksDB, Berkeley DB, Ehcache với disk store.

##### 4.9.3.1.3. Distributed Cache

- **Network-based:** Dữ liệu được phân tán trên nhiều nút trong mạng.
- **Ưu điểm:** Mở rộng dung lượng, khả năng chịu lỗi cao.
- **Nhược điểm:** Độ trễ mạng, phức tạp trong triển khai và quản lý.
- **Công nghệ phổ biến:** Redis Cluster, Hazelcast, Apache Ignite, Memcached với sharding.

#### 4.9.3.2. Cache Policies (Chính sách cache)

##### 4.9.3.2.1. Chính sách thay thế (Replacement Policies)

- **LRU (Least Recently Used)**: Loại bỏ các mục ít được sử dụng gần đây nhất.
- **LFU (Least Frequently Used)**: Loại bỏ các mục được sử dụng ít thường xuyên nhất.
- **FIFO (First In First Out)**: Loại bỏ các mục cũ nhất trước.
- **ARC (Adaptive Replacement Cache)**: Tự động cân bằng giữa recency và frequency.
- **TinyLFU**: Kết hợp frequency và recency với bộ lọc frequency hiệu quả.

##### 4.9.3.2.2. Chính sách ghi (Write Policies)

- **Write-Through**: Ghi đồng thời vào cache và storage.
- **Write-Back (Write-Behind)**: Ghi vào cache trước, sau đó mới ghi vào storage.
- **Write-Around**: Ghi trực tiếp vào storage, bỏ qua cache.

##### 4.9.3.2.3. Chính sách hết hạn (Expiration Policies)

- **TTL (Time-To-Live)**: Dữ liệu hết hạn sau một khoảng thời gian cố định kể từ khi được tạo.
- **Idle Timeout**: Dữ liệu hết hạn sau một khoảng thời gian không được truy cập.
- **Custom Expiration**: Thời gian hết hạn được xác định dựa trên loại dữ liệu hoặc metadata.

#### 4.9.3.3. Cache Synchronization (Đồng bộ hóa cache)

##### 4.9.3.3.1. Cache Invalidation (Vô hiệu hóa cache)

- **Direct Invalidation**: Xóa cache khi dữ liệu thay đổi.
- **Time-Based Invalidation**: Dữ liệu tự động hết hạn sau một khoảng thời gian.
- **Event-Based Invalidation**: Xóa cache khi có sự kiện cụ thể xảy ra.

##### 4.9.3.3.2. Cache Consistency (Tính nhất quán cache)

- **Strong Consistency**: Đảm bảo tất cả các nút nhìn thấy cùng một dữ liệu tại cùng một thời điểm.
- **Eventual Consistency**: Dữ liệu sẽ nhất quán sau một khoảng thời gian.
- **Read-Your-Writes Consistency**: Người dùng luôn thấy các thay đổi mà họ đã thực hiện.

#### 4.9.3.4. Cache Monitoring và Analytics

##### 4.9.3.4.1. Metrics cần theo dõi

- **Hit Rate**: Tỷ lệ yêu cầu được tìm thấy trong cache.
- **Miss Rate**: Tỷ lệ yêu cầu không được tìm thấy trong cache.
- **Latency**: Thời gian để truy xuất dữ liệu từ cache.

- **Eviction Rate:** Tốc độ loại bỏ dữ liệu khỏi cache.
- **Memory Usage:** Lượng bộ nhớ sử dụng của cache.

#### 4.9.3.4.2. Công cụ monitoring

- **Prometheus và Grafana:** Giám sát và hiển thị metrics.
- **Redis Stats:** Giám sát Redis cache.
- **JMX:** Giám sát cache Java.
- **Các công cụ tích hợp trong framework caching.**

### 4.9.4. Triển khai Hybrid Caching trong các ứng dụng thực tế

#### 4.9.4.1. Hybrid Caching trong ứng dụng Web

##### 4.9.4.1.1. Kiến trúc tổng thể

```
1 [Client] ↔ [CDN] ↔ [Web Server] ↔ [Local Cache] ↔ [Distributed Cache] ↔ [Database]
```

- **Browser Cache:** Cache phía client, lưu trữ static assets.
- **CDN Cache:** Cache phân tán toàn cầu cho static content.
- **Application Cache:** Local cache trong web server.
- **Distributed Cache:** Redis/Memcached cho phiên làm việc và dữ liệu chia sẻ giữa các máy chủ.
- **Database Cache:** Buffer pool, query cache.

##### 4.9.4.1.2. Ví dụ với Spring Boot và Redis

```
1 @Configuration
2 @EnableCaching
3 public class CacheConfig {
4     @Bean
5     public CacheManager cacheManager(
6         CaffeineCacheManager caffeineCacheManager,
7         RedisCacheManager redisCacheManager) {
8         return new HybridCacheManager(caffeineCacheManager,
9             redisCacheManager);
10    }
11    @Bean
12    public CaffeineCacheManager caffeineCacheManager() {
13        CaffeineCacheManager cacheManager = new
14            CaffeineCacheManager();
15        cacheManager.setCacheNames(Arrays.asList("localCache"));
16        cacheManager.setCaffeine(Caffeine.newBuilder()
17            .expireAfterWrite(60, TimeUnit.SECONDS))
```

```

17         .maximumSize(1000));
18     return cacheManager;
19 }
20
21 @Bean
22 public RedisCacheManager redisCacheManager(RedisConnectionFactory
redisConnectionFactory) {
23     RedisCacheConfiguration config =
RedisCacheConfiguration.defaultCacheConfig()
24         .entryTtl(Duration.ofMinutes(10));
25
26     return RedisCacheManager.builder(redisConnectionFactory)
27         .cacheDefaults(config)
28         .build();
29 }
30 }
31
32 // Cache manager tùy chỉnh kết hợp Caffeine (local) và Redis
(distributed)
33 public class HybridCacheManager implements CacheManager {
34     private final CaffeineCacheManager localCacheManager;
35     private final RedisCacheManager distributedCacheManager;
36
37     // ... implementation details ...
38 }

```

#### 4.9.4.2. Hybrid Caching trong Microservices

##### 4.9.4.2.1. Các thách thức trong Microservices

- **Data Consistency:** Đảm bảo dữ liệu nhất quán giữa các services.
- **Cache Invalidation:** Vô hiệu hóa cache khi dữ liệu thay đổi ở service khác.
- **Resilience:** Xử lý khi cache service gặp sự cố.

##### 4.9.4.2.2. Mẫu thiết kế

- **Local Cache + Distributed Cache:** Mỗi service có local cache, shared data được lưu trong distributed cache.
- **Cache-Aside Pattern:** Service tự quản lý việc đọc/ghi cache.
- **Event-Driven Invalidation:** Sử dụng message broker để vô hiệu hóa cache khi dữ liệu thay đổi.

##### 4.9.4.2.3. Ví dụ với Spring Cloud

```
1 @Service
```

```
Java
```

```

2  public class ProductService {
3      @Autowired
4      private ProductRepository repository;
5
6      @Autowired
7      private CacheManager localCacheManager;
8
9      @Autowired
10     private CacheManager distributedCacheManager;
11
12     // Sử dụng local cache cho dữ liệu ít thay đổi
13     @Cacheable(value = "productDetails", cacheManager =
14     "localCacheManager")
15     public ProductDetails getProductDetails(String productId) {
16         // ...
17     }
18
19     // Sử dụng distributed cache cho dữ liệu chia sẻ
20     @Cacheable(value = "productInventory", cacheManager =
21     "distributedCacheManager")
22     public ProductInventory getProductInventory(String productId) {
23         // ...
24     }
25
26     // Sử dụng messaging để vô hiệu hóa cache khi dữ liệu thay đổi
27     @StreamListener("inventoryChangeTopic")
28     public void handleInventoryChange(InventoryChangeEvent event) {
29         String cacheKey = "productInventory::" + event.getProductId();
30         distributedCacheManager.getCache("productInventory").evict(event

```

#### 4.9.5. Ưu và nhược điểm của Hybrid Caching

##### 4.9.5.1. Ưu điểm

Ưu điểm	Mô tả
<b>Hiệu suất cao</b>	Tận dụng tốc độ của cache bộ nhớ cho dữ liệu “nóng”.
<b>Khả năng mở rộng</b>	Cache trên ổ đĩa hoặc các tầng khác giúp lưu trữ được nhiều dữ liệu hơn.
<b>Tính linh hoạt</b>	Có thể tùy chỉnh chiến lược cache theo từng loại dữ liệu hoặc luồng truy cập.
<b>Giảm tải cho hệ thống gốc</b>	Giảm số lần truy vấn cơ sở dữ liệu hoặc dịch vụ phía sau.

##### 4.9.5.2. Nhược điểm

- **Tăng độ phức tạp hệ thống:** Việc đồng bộ giữa các tầng cache có thể khó kiểm soát.
- **Rủi ro về tính nhất quán dữ liệu:** Dữ liệu trong nhiều tầng có thể không đồng bộ nếu không được thiết kế cẩn thận.
- **Chi phí bảo trì cao:** Phải theo dõi, tối ưu và cập nhật các tầng cache liên tục.

#### 4.9.6. Ứng dụng của Hybrid Caching

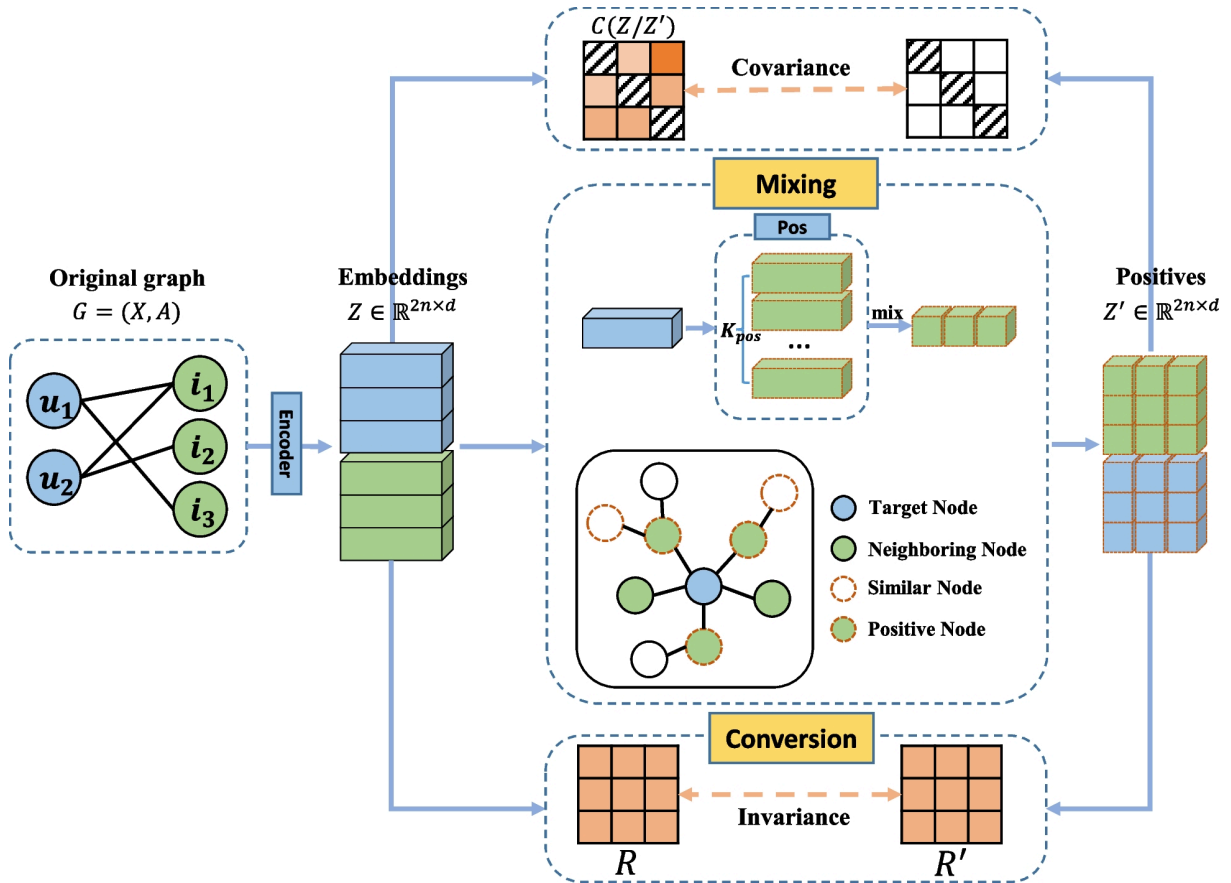
- Hybrid Caching được sử dụng rộng rãi trong các hệ thống yêu cầu hiệu suất cao:
  - **Hệ thống CDN (Content Delivery Network):** Kết hợp caching ở client, edge server và origin server.
  - **Microservices và API Gateway:** Caching phản hồi API ở nhiều tầng để giảm độ trễ.
  - **Ứng dụng thương mại điện tử:** Caching thông tin sản phẩm, giỏ hàng, đề xuất cá nhân hóa.

### 4.10. Graph Cache: Caching data in N Dimensional structures

#### 4.10.1. Giới thiệu

Graph Cache là một mô hình lưu trữ dữ liệu mới, được thiết kế để giải quyết những thách thức phức tạp trong việc lưu trữ và truy xuất dữ liệu trong các hệ thống hiện đại. Khác với các hệ thống cache truyền thống tập trung vào cặp key-value đơn giản, Graph Cache cho phép lưu trữ dữ liệu trong cấu trúc đa chiều, giúp tối ưu hóa hiệu

suất và tính linh hoạt khi xử lý dữ liệu phức tạp.



#### 4.10.2. Nguyên lý cơ bản của Graph Cache

Graph Cache tổ chức dữ liệu thành các đồ thị đa chiều, trong đó mỗi điểm (node) có thể kết nối với nhiều điểm khác thông qua các cạnh (edge). Điều này cho phép biểu diễn mối quan hệ phức tạp giữa các thực thể dữ liệu, vượt xa khả năng của các hệ thống cache truyền thống.

- Các đặc điểm chính của Graph Cache bao gồm:
  - ▶ **Cấu trúc dữ liệu đa chiều:** Dữ liệu được tổ chức trong không gian  $n$  chiều, cho phép biểu diễn và truy vấn các mối quan hệ phức tạp.
  - ▶ **Lưu trữ theo ngữ cảnh:** Graph Cache lưu trữ dữ liệu theo ngữ cảnh cụ thể, giúp tăng độ chính xác khi truy xuất.
  - ▶ **Hiệu quả trong truy vấn đa điều kiện:** Việc tìm kiếm và lọc dữ liệu dựa trên nhiều thuộc tính trở nên đơn giản và hiệu quả hơn.
  - ▶ **Khả năng mở rộng:** Cấu trúc đồ thị cho phép mở rộng dễ dàng khi thêm chiều dữ liệu mới mà không cần thay đổi lớn về thiết kế.



### 4.10.3. So sánh với các hệ thống cache truyền thống

Đặc điểm	Cache truyền thống	Graph Cache
Mô hình dữ liệu	Key-Value đơn giản	Đồ thị đa chiều
Khả năng biểu diễn quan hệ	Hạn chế	Mạnh mẽ
Truy vấn đa điều kiện	Phức tạp, kém hiệu quả	Đơn giản, hiệu quả
Mở rộng cấu trúc	Khó khăn	Linh hoạt
Quản lý bộ nhớ	Đơn giản	Phức tạp hơn

### 4.10.4. Kiến trúc của Graph Cache

Graph Cache được xây dựng từ ba thành phần chính:

#### 4.10.4.1. Data Nodes (Nút dữ liệu)

Các nút dữ liệu chứa giá trị thực tế được lưu trữ. Mỗi nút có thể chứa bất kỳ loại dữ liệu nào, từ các giá trị đơn giản đến các đối tượng phức tạp.

#### 4.10.4.2. Dimension Nodes (Nút chiều)

Các nút chiều đại diện cho các thuộc tính hoặc chiều của dữ liệu. Chúng tạo thành cấu trúc phân cấp giúp tổ chức dữ liệu theo nhiều chiều.

#### 4.10.4.3. Edges (Cạnh)

Các cạnh kết nối các nút với nhau, biểu diễn mối quan hệ giữa dữ liệu và chiều. Mỗi cạnh có thể chứa metadata bổ sung về mối quan hệ này.

### 4.10.5. Ứng dụng thực tế của Graph Cache

#### 4.10.5.1. Hệ thống thương mại điện tử

Trong hệ thống thương mại điện tử, Graph Cache có thể được sử dụng để lưu trữ thông tin sản phẩm theo nhiều chiều như danh mục, giá cả, thương hiệu, và đánh giá của người dùng. Điều này cho phép truy xuất nhanh chóng các sản phẩm dựa trên nhiều tiêu chí lọc phức tạp.

Ví dụ: Một người dùng có thể tìm kiếm “laptop dưới 20 triệu đồng, RAM 16GB, của Apple, có đánh giá 4 sao trở lên”. Graph Cache có thể truy xuất kết quả nhanh chóng bằng cách đi theo các cạnh tương ứng với từng điều kiện.

#### 4.10.5.2. Phân tích mạng xã hội

Trong các ứng dụng mạng xã hội, Graph Cache có thể lưu trữ hiệu quả mối quan hệ giữa người dùng, nội dung họ tạo ra, và tương tác của họ. Điều này giúp tối ưu hóa việc hiển thị nội dung phù hợp và phân tích xu hướng.

#### 4.10.5.3. Hệ thống gợi ý (Recommendation Systems)

Graph Cache đặc biệt phù hợp cho các hệ thống gợi ý, nơi cần phân tích mối quan hệ phức tạp giữa người dùng, sản phẩm/nội dung, và hành vi để đưa ra các đề xuất chính xác.

#### 4.10.6. Triển khai Graph Cache

Việc triển khai Graph Cache thường theo các bước sau:

- **Xác định các chiều dữ liệu:** Phân tích dữ liệu để xác định các thuộc tính quan trọng sẽ trở thành các chiều.
- **Thiết kế cấu trúc đồ thị:** Xây dựng cấu trúc nút và cạnh phù hợp với yêu cầu ứng dụng.
- **Chiến lược lưu trữ:** Quyết định cách dữ liệu được lưu trữ và cập nhật trong bộ nhớ.
- **Tối ưu hóa truy vấn:** Xây dựng các thuật toán truy vấn hiệu quả cho các trường hợp sử dụng cụ thể.
- **Quản lý bộ nhớ:** Thiết lập chiến lược xóa dữ liệu không sử dụng để giải phóng bộ nhớ.

Một ví dụ đơn giản về triển khai Graph Cache có thể sử dụng các cấu trúc dữ liệu có sẵn như sau:

```
1 public class GraphCache {
2     private Map<String, DimensionNode> dimensions;
3     private Map<String, DataNode> dataNodes;
4
5     public GraphCache() {
6         dimensions = new HashMap<>();
7         dataNodes = new HashMap<>();
8     }
9
10    public void addDimension(String name) {
11        dimensions.put(name, new DimensionNode(name));
12    }
13
14    public void addData(String id, Object value, Map<String, String>
dimensionValues) {
15        DataNode dataNode = new DataNode(id, value);
16        dataNodes.put(id, dataNode);
17
18        // Kết nối data node với các dimension node tương ứng
```

Java

```

19         for (Map.Entry<String, String> entry :
dimensionValues.entrySet()) {
20             String dimensionName = entry.getKey();
21             String dimensionValue = entry.getValue();
22
23             DimensionNode dimensionNode =
dimensions.get(dimensionName);
24             if (dimensionNode != null) {
25                 dimensionNode.addValue(dimensionValue, dataNode);
26             }
27         }
28     }
29
30     public List<Object> query(Map<String, String> criteria) {
31         // Triển khai thuật toán truy vấn dựa trên tiêu chí
32         // ...
33     }
34 }

```

#### 4.10.7. Thách thức và lưu ý khi sử dụng Graph Cache

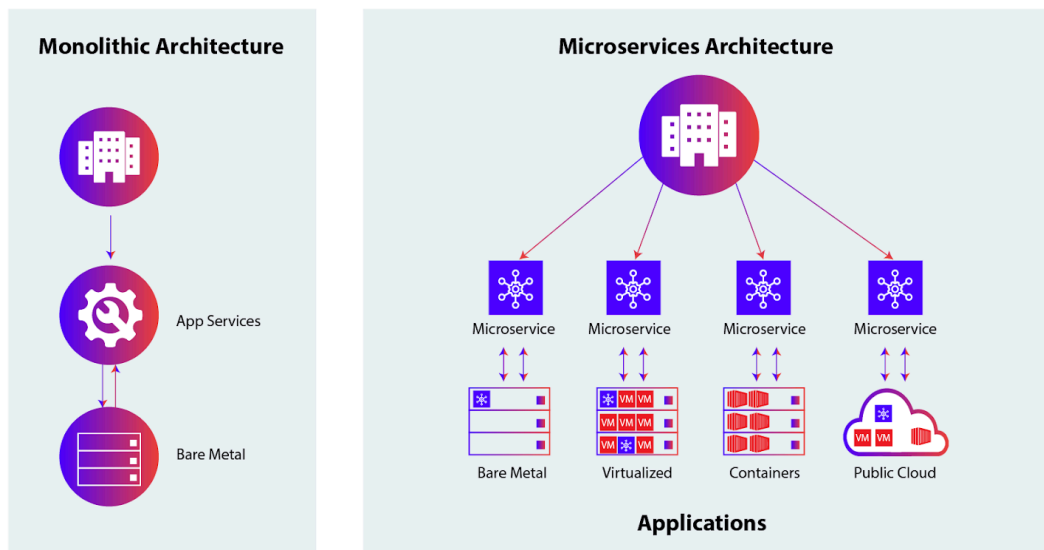
- **Quản lý bộ nhớ:** Cấu trúc đồ thị có thể tiêu thụ nhiều bộ nhớ hơn so với các cấu trúc cache đơn giản. Cần có chiến lược xóa dữ liệu hiệu quả.
- **Độ phức tạp thuật toán:** Các thuật toán duyệt đồ thị có thể phức tạp và tốn kém về mặt tính toán nếu không được tối ưu hóa.
- **Đồng bộ hóa:** Trong môi trường đa luồng, việc duy trì tính nhất quán của đồ thị có thể là thách thức.
- **Tính toàn vẹn dữ liệu:** Cần có cơ chế xử lý các tham chiếu không hợp lệ hoặc vòng lặp trong đồ thị.

## Chương 5. Áp dụng cache vào microservice

### 5.1. Giới thiệu microservice

- Kiến trúc microservice là một phương pháp phát triển phần mềm trong đó ứng dụng được xây dựng như một tập hợp các dịch vụ nhỏ, độc lập, mỗi dịch vụ chạy trong

quy trình riêng biệt và giao tiếp thông qua các cơ chế nhẹ, thường là API HTTP.



#### 5.1.1. Đặc điểm chính của Microservice

- **Phân tách theo chức năng:** Mỗi dịch vụ đại diện cho một khả năng kinh doanh cụ thể.
- **Độc lập:** Các dịch vụ có thể được phát triển, triển khai và mở rộng độc lập.
- **Công nghệ đa dạng:** Các nhóm có thể chọn công nghệ phù hợp nhất cho mỗi dịch vụ.
- **Quản lý dữ liệu phân tán:** Mỗi dịch vụ quản lý cơ sở dữ liệu riêng.
- **Khả năng chịu lỗi:** Lỗi trong một dịch vụ không ảnh hưởng đến toàn bộ hệ thống.

#### 5.1.2. Lợi ích

- **Triển khai nhanh chóng:** Phát triển và triển khai dịch vụ nhanh hơn do kích thước nhỏ.
- **Khả năng mở rộng linh hoạt:** Các dịch vụ có thể được mở rộng riêng biệt.
- **Dễ hiểu và bảo trì:** Mã nguồn đơn giản hơn trong mỗi dịch vụ.
- **Hỗ trợ làm việc song song:** Các nhóm có thể phát triển các dịch vụ khác nhau cùng lúc.

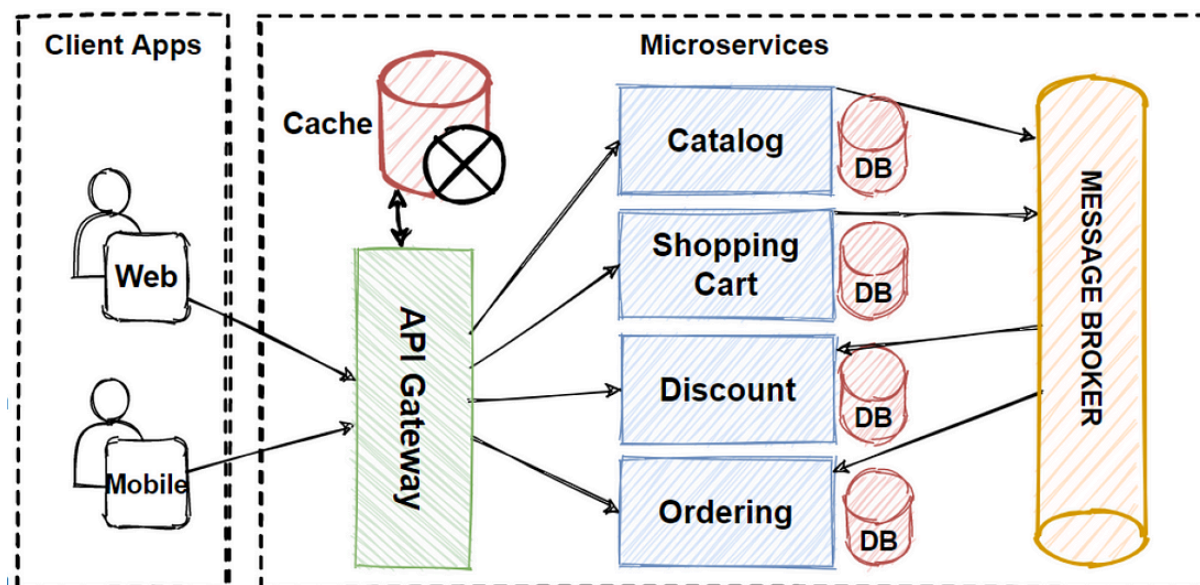
#### 5.1.3. Thách thức

- **Độ phức tạp phân tán:** Quản lý hệ thống phân tán khó khăn hơn.
- **Giao tiếp giữa các dịch vụ:** Cần thiết kế giao tiếp hiệu quả.
- **Tính nhất quán dữ liệu:** Khó đảm bảo trong môi trường phân tán.
- **Giám sát và gỡ lỗi:** Yêu cầu công cụ phức tạp hơn.

## 5.2. Distributed cache

### 5.2.1. Giới thiệu

Trong kiến trúc **Microservices**, việc giao tiếp giữa các dịch vụ và truy cập dữ liệu phân tán là điều không thể tránh khỏi. Một trong những thách thức lớn là hiệu năng truy vấn dữ liệu và giảm tải cho hệ thống backend (cơ sở dữ liệu, API bên thứ ba...). **Distributed Cache** (bộ nhớ đệm phân tán) là giải pháp quan trọng giúp giải quyết vấn đề này bằng cách lưu trữ tạm thời dữ liệu gần nơi sử dụng, từ đó tăng tốc độ truy cập và cải thiện hiệu năng toàn hệ thống.



### 5.2.2. Định nghĩa Distributed Cache

- **Distributed Cache** là một hệ thống bộ nhớ đệm được chia sẻ giữa nhiều node (máy chủ), cho phép nhiều ứng dụng hoặc dịch vụ cùng truy cập, đọc và ghi dữ liệu một cách nhất quán và nhanh chóng. Khác với **Local Cache** (chỉ tồn tại trong một instance), Distributed Cache hoạt động như một dịch vụ độc lập.

#### 5.2.2.1. Ví dụ các hệ thống Distributed Cache phổ biến:

- Redis
- Memcached
- Hazelcast
- Apache Ignite

### 5.2.3. Lợi ích của Distributed Cache trong Microservices

Lợi ích	Mô tả
<b>Tăng hiệu năng</b>	Truy xuất dữ liệu từ cache nhanh hơn nhiều so với truy vấn cơ sở dữ liệu.
<b>Giảm tải cho Database</b>	Hạn chế số lượng truy vấn trực tiếp đến DB.
<b>Tính nhất quán cao hơn Local Cache</b>	Dữ liệu được chia sẻ giữa các service, tránh hiện tượng “cache lệch”.
<b>Tăng khả năng mở rộng (Scalability)</b>	Các node microservice có thể scale ngang mà vẫn truy cập được cache chung.
<b>Hỗ trợ failover và replication</b>	Nhiều hệ thống cache hỗ trợ cơ chế tự phục hồi và dự phòng.

### 5.2.4. Kiến trúc Tích hợp Distributed Cache

Một số mô hình phổ biến trong tích hợp Distributed Cache vào microservices:

#### 5.2.4.1. Cache-Aside (Lazy Loading)

- Ứng dụng truy vấn cache trước, nếu không có dữ liệu (cache miss), sẽ truy cập DB và sau đó ghi lại vào cache.
- Phù hợp với dữ liệu thường xuyên thay đổi hoặc có yêu cầu consistency cao.

#### 5.2.4.2. Write-Through / Write-Behind

- Khi ghi dữ liệu vào DB, hệ thống cũng ghi đồng thời vào cache (write-through) hoặc ghi cache trước rồi đồng bộ ra DB sau (write-behind).
- Phù hợp với hệ thống yêu cầu ghi dữ liệu nhanh hoặc có độ trễ ghi vào DB chấp nhận được.

### 5.2.5. Vấn đề & Thách thức

Thách thức	Mô tả
<b>Consistency</b>	Dữ liệu cache và DB có thể không đồng bộ nếu không kiểm soát tốt.
<b>Cache Invalidation</b>	Xóa cache đúng lúc là khó: không nên xóa quá sớm hoặc quá muộn.
<b>Split-brain / Data inconsistency</b>	Xảy ra khi hệ thống phân tán bị chia tách mạng (network partition).
<b>Quản lý TTL (Time-To-Live)</b>	Nếu TTL quá thấp, hệ thống sẽ liên tục cache miss; quá cao thì dữ liệu cũ tồn tại quá lâu.
<b>Chi phí vận hành</b>	Redis/Memcached cluster yêu cầu hệ thống monitoring, backup, scaling,...

### 5.2.6. Best Practices

- Sử dụng **Redis Cluster** để đảm bảo High Availability.
- Sử dụng **Cache Key có tiền tố (prefix)** để tránh trùng lặp giữa các microservice.
- Kết hợp với **Circuit Breaker** để tránh cache downtime ảnh hưởng hệ thống chính.
- Định nghĩa rõ **TTL** cho từng loại dữ liệu.
- Định kỳ thực hiện **cache warm-up** sau khi deploy hệ thống.

## 5.3. Event-Driven Cache Invalidation

### 5.3.1. 1. Giới thiệu

Trong kiến trúc microservice, mỗi dịch vụ thường có cơ sở dữ liệu riêng biệt để đảm bảo tính độc lập và phân tách trách nhiệm. Để tối ưu hiệu suất, các hệ thống thường sử dụng cache (bộ nhớ đệm) để lưu trữ tạm thời dữ liệu truy xuất nhiều lần. Tuy nhiên, một thách thức lớn đặt ra là: **Làm thế nào để đảm bảo dữ liệu trong cache luôn đồng bộ với dữ liệu thực trong cơ sở dữ liệu, đặc biệt là khi nhiều service cùng chia sẻ một phần dữ liệu?**

Một giải pháp phổ biến và hiệu quả là **Event-Driven Cache Invalidation (Hủy cache dựa trên sự kiện)**.

### 5.3.2. Vấn đề của Cache trong Microservice

Giả sử hệ thống có hai service:

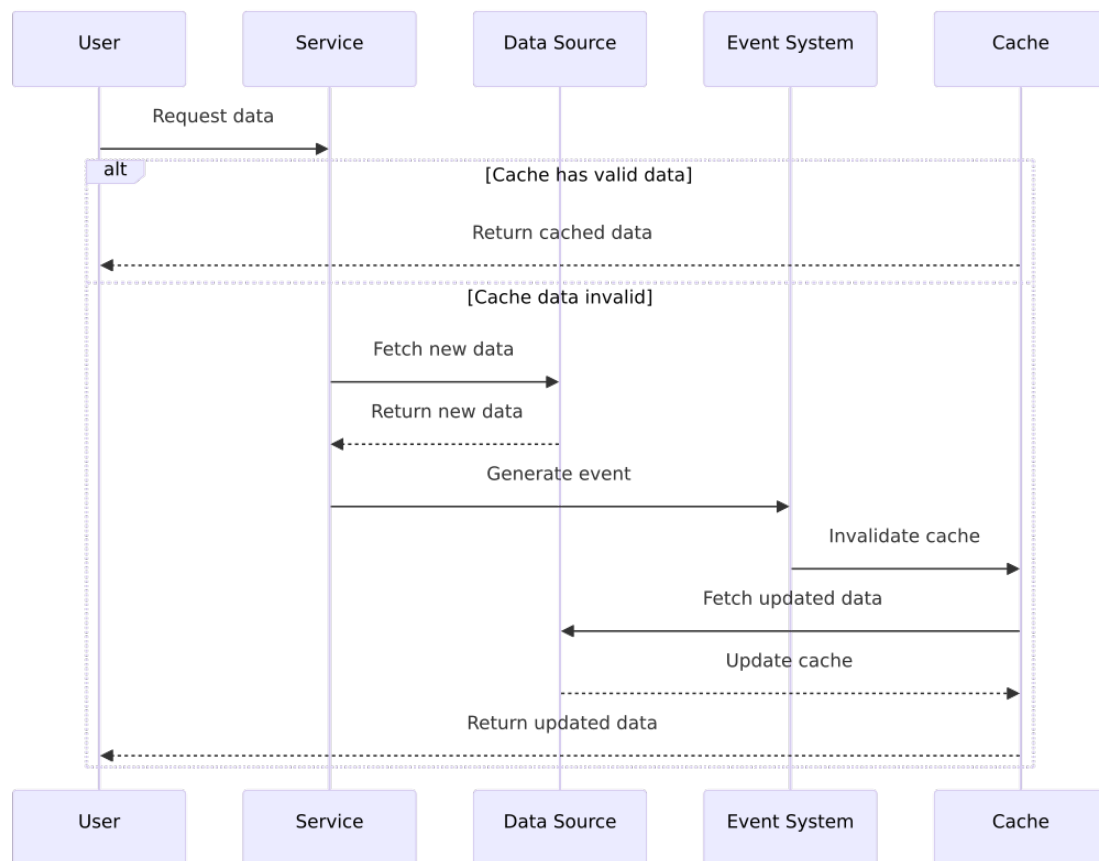
- **User Service**: quản lý thông tin người dùng.
- **Order Service**: xử lý đơn hàng, có lưu thông tin người dùng liên quan đến đơn.

Khi một người dùng thay đổi thông tin cá nhân (ví dụ tên hoặc địa chỉ), **User Service** cập nhật cơ sở dữ liệu của nó, nhưng dữ liệu người dùng được cache ở **Order Service** lại không được tự động cập nhật. Điều này gây ra hiện tượng **cache stale** (cache lỗi thời), dẫn đến việc trả về thông tin không chính xác.

### 5.3.3. Event-Driven Cache Invalidation là gì?

**Event-Driven Cache Invalidation** là một phương pháp mà trong đó, khi dữ liệu được thay đổi ở một service (source of truth), service đó sẽ **phát ra một sự kiện (event)** thông báo rằng dữ liệu đã thay đổi. Các service khác, nếu cache dữ liệu liên quan, sẽ **nghe (subscribe)** sự kiện này và thực hiện **hủy cache (invalidate)** hoặc cập nhật lại

dữ liệu trong cache.



#### 5.3.4. Quy trình hoạt động

- **User Service** cập nhật thông tin người dùng trong cơ sở dữ liệu.
- Nó phát ra sự kiện `UserUpdated` với payload chứa ID người dùng và các trường dữ liệu bị thay đổi.
- **Order Service**, vốn đang cache dữ liệu người dùng, **nghe sự kiện `UserUpdated`** này.
- Khi nhận được sự kiện, **Order Service xóa cache** dữ liệu cũ hoặc cập nhật lại cache bằng cách gọi API hoặc chờ một sync event khác.

#### 5.3.5. Ưu điểm

- **Tính nhất quán cao hơn**: Cache luôn được cập nhật theo thời gian thực hoặc gần thời gian thực.
- **Hiệu suất tốt hơn so với polling**: Thay vì liên tục kiểm tra sự thay đổi, chỉ cần phản ứng khi có sự kiện.
- **Phù hợp với kiến trúc microservice**: Mỗi service độc lập, giao tiếp thông qua các sự kiện.

#### 5.3.6. Nhược điểm và thách thức

- **Độ phức tạp tăng**: Phải thiết kế hệ thống event bus hoặc message broker (như Kafka, RabbitMQ, NATS).
- **Khó debug hơn**: Các luồng dữ liệu không còn tuyến tính.



- **Xử lý lỗi cần cẩn thận:** Trường hợp service nhận sự kiện bị down, hoặc mất sự kiện, có thể gây lỗi dữ liệu.

### 5.3.7. Mô hình triển khai phổ biến

#### 5.3.7.1. Sơ đồ tổng quát:

```

1  [User Service]
2      |
3      |-- Update User
4      |-- Emit "UserUpdated" Event
5      |
6  [Message Broker] → Kafka / RabbitMQ / NATS
7      |
8  [Order Service] (Subscriber)
9      |
10     |-- Invalidate Cache
11     |-- Optionally fetch fresh data

```

### 5.3.8. Ví dụ với Redis Cache và Kafka

- **Cache Engine:** Redis
- **Event System:** Kafka
- **Luồng sự kiện:**
  - User Service gửi event `user.updated` đến Kafka.
  - Order Service có Redis cache dạng key: `user:{id}`.
  - Khi nhận được `user.updated`, nó gọi `redis.del("user:{id}")` để xóa cache.

## 5.4. Distributed locking

### 5.4.1. Giới thiệu

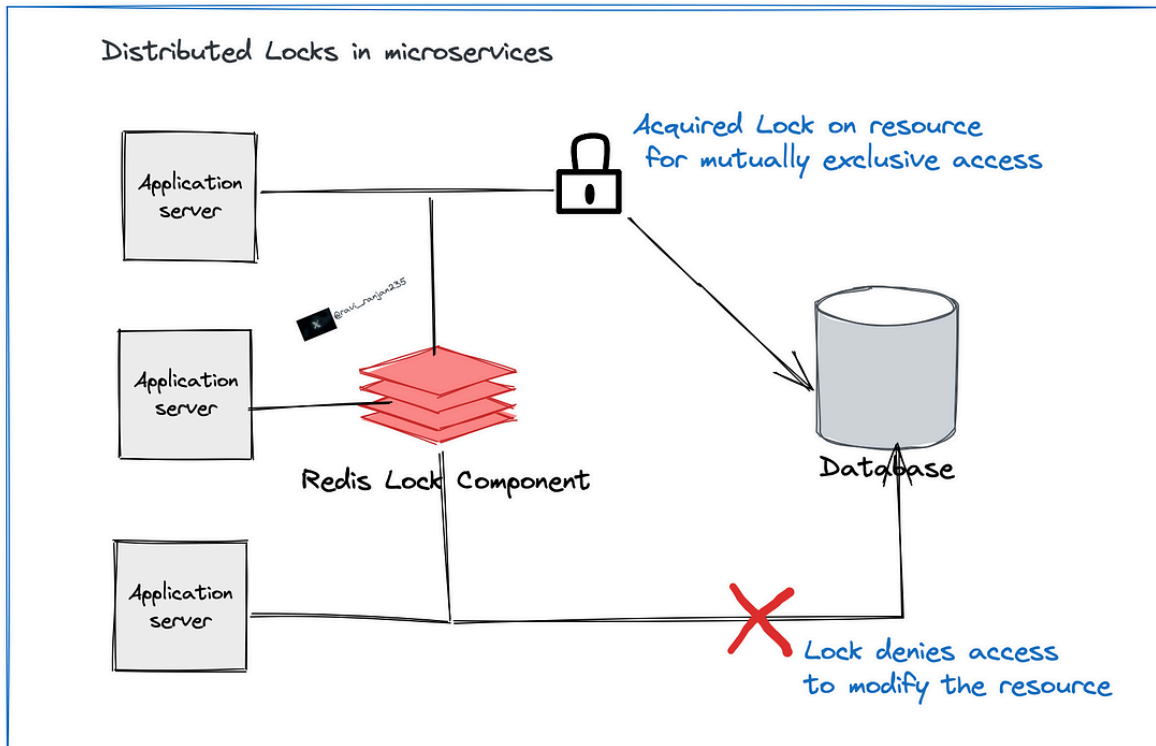
Trong kiến trúc microservices, các dịch vụ nhỏ hoạt động độc lập và giao tiếp với nhau qua mạng. Mỗi dịch vụ có thể có cơ sở dữ liệu riêng, triển khai riêng biệt và được mở rộng độc lập. Tuy nhiên, điều này cũng tạo ra một số thách thức khi nhiều service cùng truy cập hoặc thay đổi dữ liệu chia sẻ. **Distributed Locking** (khóa phân tán) là một kỹ thuật được sử dụng để đảm bảo **tính nhất quán dữ liệu** và **tránh xung đột truy cập** trong môi trường phân tán.

### 5.4.2. Vấn đề đặt ra

Khi nhiều instance của một service hoặc nhiều service khác nhau cố gắng truy cập và ghi vào cùng một tài nguyên (ví dụ: cập nhật tồn kho sản phẩm), có thể xảy ra race condition, gây ra sai lệch dữ liệu. Trong môi trường đơn lẻ, ta có thể sử dụng `synchronized`, `mutex`, hoặc `lock`. Tuy nhiên, trong microservices, ta cần một cơ chế khóa hoạt động **trên nhiều máy chủ, đồng bộ qua mạng**.

### 5.4.3. Định nghĩa Distributed Lock

- **Distributed Lock** là một cơ chế giúp các hệ thống phân tán (chạy trên nhiều node) có thể **đồng bộ hóa truy cập đến tài nguyên dùng chung** bằng cách đảm bảo chỉ có một node tại một thời điểm được phép nắm giữ “quyền truy cập” đến tài nguyên đó.
- Ví dụ, nếu một service A đang thực hiện việc cập nhật giá trị đơn hàng, Distributed Lock đảm bảo rằng các service khác sẽ phải chờ đến khi service A hoàn tất.



### 5.4.4. Một số cách triển khai Distributed Lock

#### 5.4.4.1. Dùng Redis (Redlock Algorithm)

- Redis là một key-value store có tốc độ cao và được dùng rộng rãi để triển khai distributed lock.
- **Redlock** là thuật toán do chính tác giả Redis đề xuất.
- Redis lock thường sử dụng lệnh SET key value NX PX time.
- Ưu điểm:
  - Hiệu năng cao
  - Dễ tích hợp
- Nhược điểm:
  - Phụ thuộc vào Redis (phải đảm bảo độ tin cậy của Redis cluster)
  - Nếu không dùng đúng cách có thể dẫn đến lock bị mất (ví dụ khi service bị crash)

#### 5.4.4.2. Dùng Zookeeper

- Zookeeper dùng node (znode) để tạo các lock theo dạng hàng đợi.

- Service tạo một znode tạm thời và theo dõi thứ tự để biết khi nào đến lượt mình thực thi.
- Ưu điểm:
  - Đảm bảo tính nhất quán mạnh
- Nhược điểm:
  - Phức tạp hơn Redis
  - Cần triển khai và duy trì Zookeeper cluster

#### 5.4.4.3. Dùng cơ chế cơ sở dữ liệu

- Tạo bảng lock trong database, sử dụng cơ chế SELECT FOR UPDATE hoặc cờ trạng thái để kiểm soát quyền truy cập.
- Ưu điểm:
  - Dễ triển khai nếu hệ thống đã có DB dùng chung
- Nhược điểm:
  - Không thực sự “phân tán” nếu chỉ dùng 1 database
  - Hiệu năng thấp hơn Redis hoặc Zookeeper

#### 5.4.5. Một số thư viện hỗ trợ phổ biến

- **Redisson** (Redis-based lock cho Java)
- **Spring Cloud Distributed Lock** (thường kết hợp Redis, Zookeeper, Consul)
- **Apache Curator** (cho Zookeeper)
- **Etd** cũng hỗ trợ lock (ít phổ biến hơn ở VN)

#### 5.4.6. Khi nào nên dùng Distributed Lock

Distributed Lock nên dùng khi:

- Có cùng một tài nguyên được nhiều service cùng truy cập/ghi đồng thời.
- Cần đảm bảo tính nguyên tử của một hành động kéo dài qua nhiều bước.
- Các thao tác có rủi ro race condition và không thể giải quyết bằng eventual consistency.

Tuy nhiên, không nên lạm dụng distributed lock vì:

- Làm tăng độ phức tạp hệ thống
- Làm giảm hiệu năng (blocking, retry, timeout...)
- Có thể tạo deadlock nếu thiết kế không cẩn thận

### 5.5. Cache access management (phân quyền giống như csdl)

#### 5.5.1. Giới thiệu

Khi các microservice trong hệ thống phân tán cùng chia sẻ tài nguyên cache, nếu không có cơ chế phân quyền phù hợp, nhiều vấn đề có thể phát sinh:

- **Rủi ro bảo mật dữ liệu:** Microservice không được phép có thể truy cập dữ liệu nhạy cảm được lưu trong cache.
- **Xung đột dữ liệu:** Các microservice có thể ghi đè lên dữ liệu của nhau nếu không kiểm soát quyền ghi/đọc.
- **Hiệu suất không đồng đều:** Một microservice có thể chiếm dụng tài nguyên cache, ảnh hưởng đến các service khác.
- **Khó khăn trong quản lý:** Không có cách nào để theo dõi và kiểm soát việc sử dụng cache của từng microservice.

## 5.5.2. Mô hình phân quyền truy cập cache

### 5.5.2.1. Phân vùng namespace

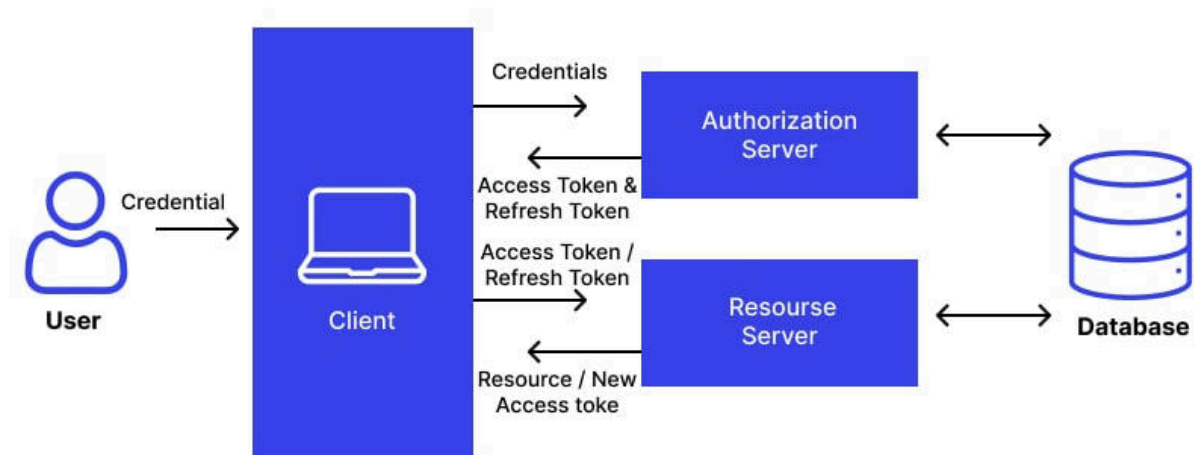
Một phương pháp cơ bản là phân chia cache thành các namespace riêng biệt cho từng microservice. Tương tự như schema trong cơ sở dữ liệu, mỗi namespace có thể được gán cho một microservice cụ thể, giới hạn phạm vi truy cập dữ liệu.

```
1 // Ví dụ sử dụng Redis với namespace
2 userService.set("user:namespace:userId123", userData)
3 orderService.set("order:namespace:orderId456", orderData)
```

Cách tiếp cận này đơn giản nhưng dựa vào sự tuân thủ quy tắc của các microservice, thiếu cơ chế thực thi nghiêm ngặt.

### 5.5.2.2. Token-based authentication

Sử dụng token xác thực để kiểm soát quyền truy cập vào cache. Mỗi microservice được cấp một token với các quyền cụ thể, và hệ thống cache kiểm tra token này trước khi cho phép truy cập.



```

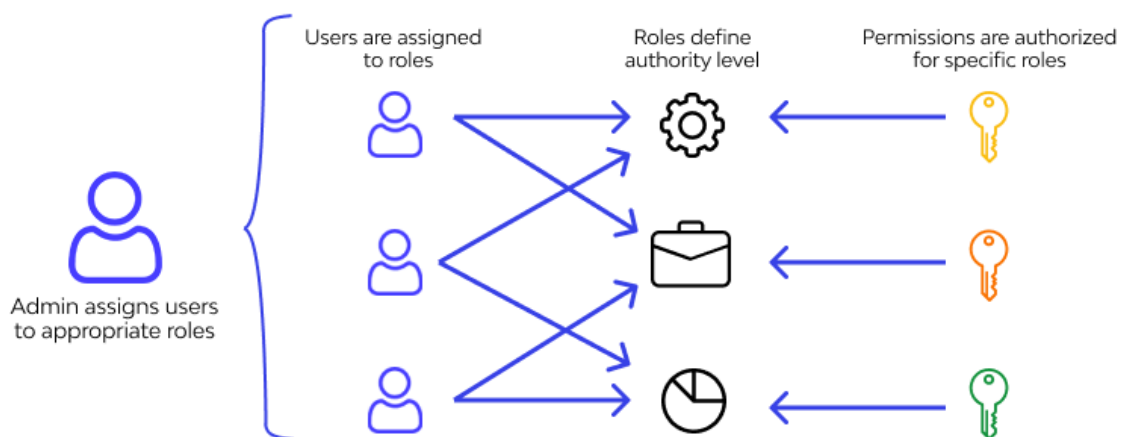
1 // Ví dụ phía cache proxy
2 public CacheResponse processRequest(CacheRequest request) {
3     String token = request.getToken();
4     String resource = request.getResource();
5
6     if (!authService.validateAccess(token, resource,
7         request.getOperation())) {
8         return CacheResponse.forbidden();
9     }
10
11     // Thực hiện thao tác cache nếu được phép
12     return cacheProvider.execute(request);
13 }

```

### 5.5.2.3. Role-Based Access Control (RBAC)

Áp dụng mô hình RBAC tương tự như trong cơ sở dữ liệu, nơi mỗi microservice được gán các vai trò (role) và mỗi vai trò có các quyền cụ thể đối với các dữ liệu cache.

#### Role-Based Access Control



```

1 // Cấu hình RBAC cho cache
2 roles:
3   - name: paymentService
4     permissions:
5       - resource: "payment:*"
6         operations: [read, write, delete]
7       - resource: "user:*"

```

```

8         operations: [read]
9
10    - name: userService
11      permissions:
12        - resource: "user:*"
13          operations: [read, write, delete]

```


#### 5.5.2.4. Attribute-Based Access Control (ABAC)

ABAC cung cấp mô hình phân quyền chi tiết hơn dựa trên thuộc tính của đối tượng, hành động và môi trường. Điều này cho phép các quy tắc phân quyền phức tạp như “dịch vụ thanh toán chỉ có thể đọc dữ liệu người dùng trong giờ làm việc”.

```

1  // Ví dụ quy tắc ABAC
2  Rule {
3      Subject: "paymentService"
4      Resource: "user:financial:*"
5      Action: "read"
6      Environment: {
7          time: "8:00-17:00",
8          dayOfWeek: "Monday-Friday"
9      }
10     Effect: PERMIT
11 }

```

 Java

### 5.5.3. Triển khai phân quyền cache


#### 5.5.3.1. Cache Proxy Pattern

Triển khai một lớp proxy trung gian giữa các microservice và hệ thống cache. Proxy này chịu trách nhiệm kiểm tra quyền truy cập trước khi chuyển tiếp yêu cầu đến cache.

```

1  @Service
2  public class SecuredCacheService implements CacheService {
3      private final RawCacheProvider cacheProvider;
4      private final AuthorizationService authService;
5
6      @Override
7      public Object get(String key, String serviceId, String token) {
8          if (!authService.canAccess(serviceId, token, key, "read")) {
9              throw new AccessDeniedException("No permission to read: "
10                 + key);
11          }
12          return cacheProvider.get(key);
13      }
14  }

```

 Java

```

12     }
13
14     @Override
15     public void set(String key, Object value, String serviceId, String
token) {
16         if (!authService.canAccess(serviceId, token, key, "write")) {
17             throw new AccessDeniedException("No permission to write: "
+ key);
18         }
19         cacheProvider.set(key, value);
20     }
21 }

```

### 5.5.3.2. Tích hợp với API Gateway

Sử dụng API Gateway như một điểm kiểm soát trung tâm cho các yêu cầu truy cập cache. Gateway có thể xác thực và phân quyền truy cập cache cho từng microservice.

```

1  // Cấu hình trong API Gateway
2  routes:
3    - path: /cache/**
4      service: cache-service
5      security:
6        authenticationRequired: true
7        permissions:
8          - role: payment-service
9            resources: ["/cache/payments/**", "/cache/users/*/payment-
info"]
10         methods: [GET, PUT]

```

### 5.5.3.3. Sidecar Pattern

Triển khai một sidecar container kèm theo mỗi microservice, hoạt động như một proxy cục bộ cho cache. Sidecar này đảm bảo tất cả các truy cập cache từ microservice đều được kiểm tra quyền.

```

1  # Kubernetes config với sidecar pattern
2  apiVersion: v1
3  kind: Pod
4  metadata:
5    name: payment-service
6  spec:
7    containers:
8      - name: payment-app

```

YAML

```

9     image: payment-service:1.0
10   - name: cache-sidecar
11     image: cache-sidecar:1.0
12     env:
13       - name: SERVICE_ID
14         value: "payment-service"
15       - name: PERMISSIONS_CONFIG
16         value: "/etc/cache-permissions/config.yaml"

```

#### 5.5.3.4. Middleware trong Client Library

Xây dựng các thư viện client cache có tích hợp sẵn middleware phân quyền, giúp các microservice tuân thủ cơ chế phân quyền mà không cần thêm code phức tạp.

```

1 CacheClient client = CacheClientBuilder.newBuilder()
2   .withEndpoint("cache.example.com")
3   .withServiceCredentials("payment-service", "api-key-xyz")
4   .withMiddleware(new AuthorizationMiddleware())
5   .build();
6
7 // Middleware tự động xử lý phân quyền
8 client.get("user:123:payment-info"); // Được phép
9 client.get("user:123:medical-history"); // Ném AccessDeniedException

```

#### 5.5.4. Các công nghệ và công cụ hỗ trợ

##### 5.5.4.1. Redis ACL (Access Control List)

Redis từ phiên bản 6.0 đã hỗ trợ ACL, cho phép kiểm soát chi tiết quyền truy cập vào các key và lệnh:

```

1 # Cấu hình Redis ACL
2 user payment-service on >payment-secret ~payment:* ~user*:payment
3   +@read +@write -@admin
4 user user-service on >user-secret ~user:* +@all -@admin

```

##### 5.5.4.2. AWS ElastiCache IAM Authentication

Với AWS ElastiCache, có thể sử dụng IAM để phân quyền truy cập:

```

1 {
2   "Version": "2012-10-17",
3   "Statement": [
4     {
5       "Effect": "Allow",

```



```

6      "Action": [
7          "elasticache:Connect"
8      ],
9      "Resource": [
10         "arn:aws:elasticache:us-east-1:account-id:cluster:my-cache/
          payment-namespace"
11     ]
12 }
13 ]
14 }

```

#### 5.5.4.3. HashiCorp Vault

Sử dụng Vault để quản lý các thông tin xác thực và token truy cập cache:

```

1  # Tạo token với quyền đọc ghi cho service payment
2  vault write auth/token/create policies=payment-cache-policy
3
4  # Cấu hình policy
5  path "cache/payment/*" {
6      capabilities = ["read", "write", "delete"]
7  }
8
9  path "cache/user/*/payment-info" {
10     capabilities = ["read"]
11 }

```

#### 5.5.4.4. OPA (Open Policy Agent)

OPA cung cấp giải pháp ủy quyền tập trung có thể áp dụng cho cache:

```

1  # Policy cho truy cập cache
2  package cache.authz
3
4  default allow = false
5
6  allow {
7      input.service == "payment-service"
8      input.method == "GET"
9      startswith(input.key, "payment:")
10 }
11
12 allow {
13     input.service == "payment-service"

```

```
14     input.method == "GET"
15     startswith(input.key, "user:")
16     endswith(input.key, ":payment-info")
17 }
```

### 5.5.5. Một số thách thức và giải pháp trong các khía cạnh

#### 5.5.5.1. Hiệu suất và độ trễ

**Thách thức:** Việc kiểm tra quyền truy cập thêm độ trễ vào mỗi thao tác cache.

**Giải pháp:**

- Cache quyết định phân quyền
- Sử dụng in-memory verification
- Thiết kế quy tắc phân quyền đơn giản, dễ kiểm tra

#### 5.5.5.2. Vấn đề nhất quán trong phân tán

**Thách thức:** Trong hệ thống phân tán, việc cập nhật và áp dụng thay đổi chính sách phân quyền cần được đồng bộ.

**Giải pháp:**

- Sử dụng hệ thống quản lý cấu hình trung tâm như etcd, Consul
- Thiết kế cơ chế phân phối cập nhật chính sách đồng bộ
- Triển khai hệ thống monitoring để phát hiện sự không nhất quán

#### 5.5.5.3. Cache invalidation và quyền xóa

**Thách thức:** Xác định microservice nào có quyền invalidate cache entries.

**Giải pháp:**

- Thiết kế cơ chế “ownership” cho các cache keys
- Áp dụng mô hình phân quyền chi tiết cho thao tác invalidation
- Sử dụng publish-subscribe để thông báo invalidation

#### 5.5.5.4. Debugging và troubleshooting

**Thách thức:** Khi gặp vấn đề truy cập cache, khó xác định là do lỗi phân quyền hay lỗi kỹ thuật.

**Giải pháp:**

- Logging chi tiết các quyết định phân quyền
- Xây dựng công cụ kiểm tra quyền (permission testing tool)
- Tích hợp hệ thống quản lý phân quyền với monitoring stack

## 5.6. Scale cache (sharding, master-slave,...)

### 5.6.1. Giới thiệu

Trong kiến trúc microservice, mỗi service hoạt động độc lập và có thể được triển khai, mở rộng và quản lý riêng biệt. Cache giữ vai trò then chốt trong việc:

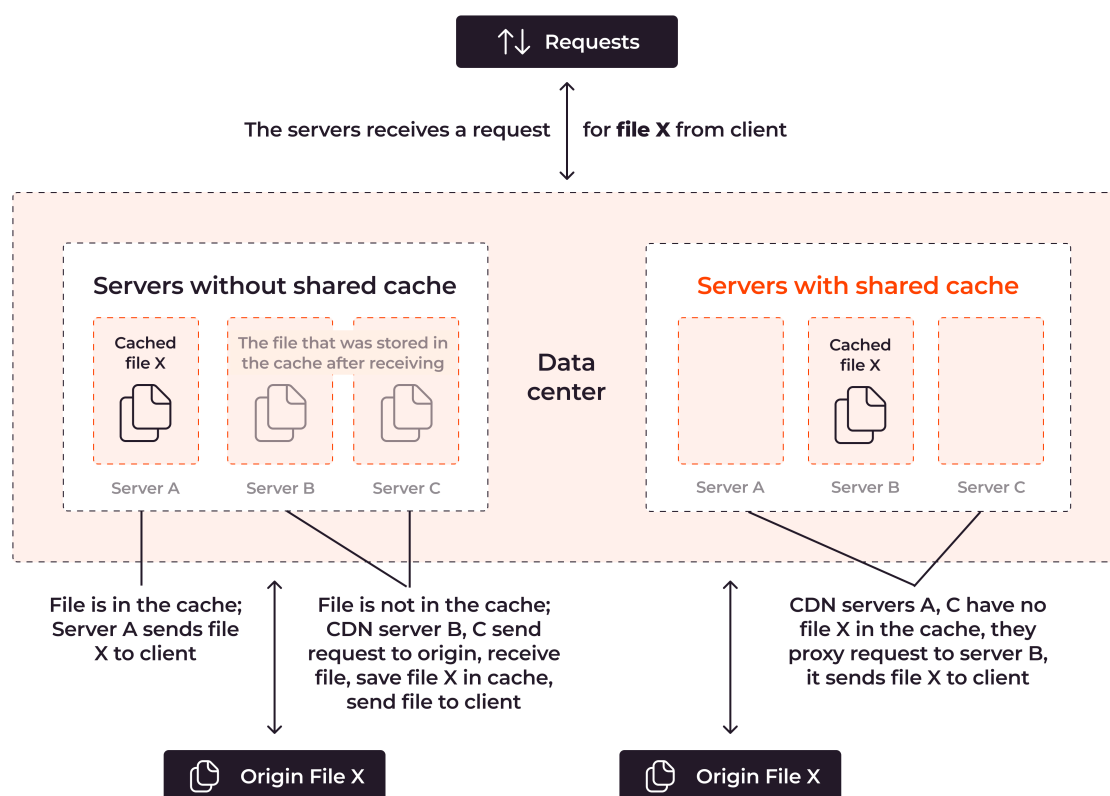
- Giảm thiểu độ trễ khi truy xuất dữ liệu
- Giảm tải cho cơ sở dữ liệu chính
- Tăng khả năng xử lý đồng thời
- Cải thiện trải nghiệm người dùng thông qua thời gian phản hồi nhanh hơn

Tuy nhiên, khi hệ thống mở rộng quy mô, việc quản lý cache trở nên phức tạp hơn, đòi hỏi các chiến lược tinh vi để đảm bảo tính nhất quán, sẵn sàng và hiệu suất cao.

### 5.6.2. Các Chiến Lược Scale Cache trong Microservice

#### 5.6.2.1. Cache Sharding (Phân mảnh Cache)

Cache sharding là kỹ thuật phân chia dữ liệu cache thành nhiều phần nhỏ hơn và phân phối chúng trên nhiều node khác nhau. Mỗi node chịu trách nhiệm lưu trữ và phục vụ một phần dữ liệu cụ thể.



- Phương pháp sharding phổ biến:

- ▶ **Hash-based Sharding:** Sử dụng hàm băm để xác định node nào sẽ lưu trữ một mục dữ liệu cụ thể. Ví dụ, với key “user\_123”, hệ thống áp dụng hàm băm và mod với số lượng node để xác định vị trí lưu trữ.
- ▶ **Range-based Sharding:** Phân chia dữ liệu dựa trên phạm vi giá trị. Ví dụ, node 1 lưu trữ người dùng có ID từ 1-1000, node 2 lưu trữ ID từ 1001-2000, v.v.
- ▶ **Directory-based Sharding:** Sử dụng bảng tra cứu để ánh xạ dữ liệu với node lưu trữ, cho phép phân phối động và linh hoạt hơn.
- **Ưu điểm của Cache Sharding:**
  - ▶ Mở rộng theo chiều ngang dễ dàng
  - ▶ Tăng dung lượng lưu trữ tổng thể
  - ▶ Giảm áp lực cho mỗi node riêng lẻ
  - ▶ Cải thiện thông lượng hệ thống
- **Thách thức:**
  - ▶ Quản lý phân phối dữ liệu không đồng đều (hot spots)
  - ▶ Xử lý việc thêm hoặc xóa node (re-sharding)
  - ▶ Đảm bảo tính nhất quán giữa các shard

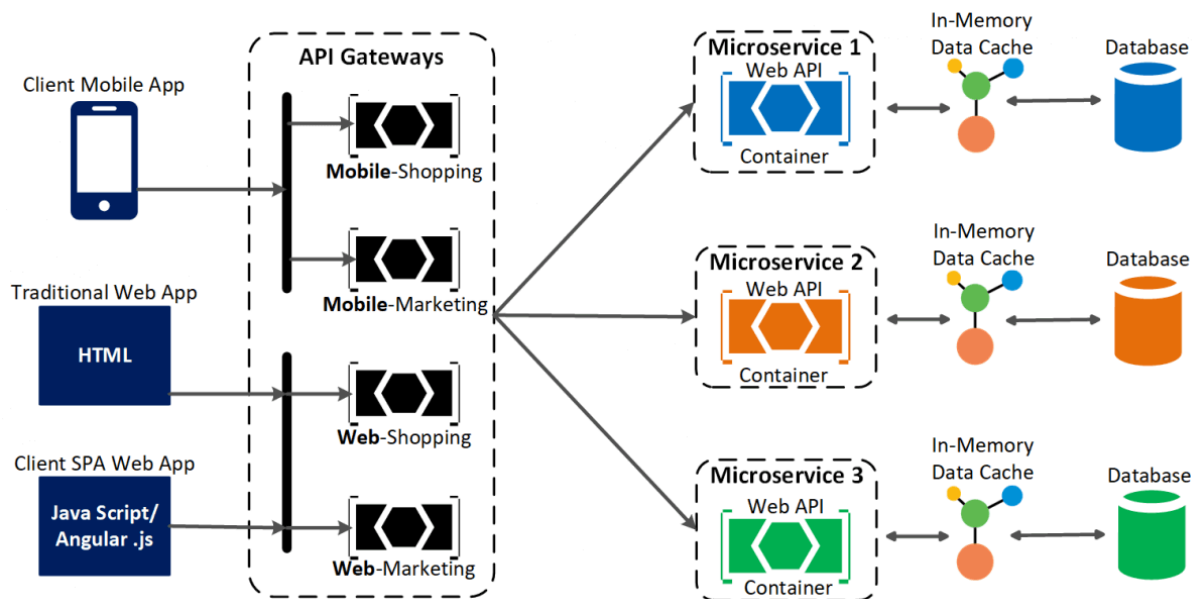
#### 5.6.2.2. Master-Slave Replication

Mô hình master-slave áp dụng cho cache sử dụng một node chính (master) để xử lý tất cả các thao tác ghi, trong khi các node phụ (slave) sao chép dữ liệu từ master và xử lý các thao tác đọc.

- **Cách thức hoạt động:**
  - Mọi thay đổi cache (ghi, cập nhật, xóa) được thực hiện trên node master
  - Master sao chép thay đổi đến tất cả các slave
  - Các yêu cầu đọc được phân phối giữa các slave để cân bằng tải
- **Ưu điểm:**
  - ▶ Tăng khả năng đọc dữ liệu
  - ▶ Cung cấp dự phòng trong trường hợp một node gặp sự cố
  - ▶ Phân tách nghiệp vụ đọc và ghi
- **Thách thức:**
  - ▶ Độ trễ sao chép có thể dẫn đến dữ liệu không nhất quán tạm thời
  - ▶ Điểm lỗi duy nhất ở node master
  - ▶ Phức tạp khi xử lý chuyển đổi dự phòng (failover)

### 5.6.2.3. Distributed Cache Clusters

Các giải pháp cache phân tán như Redis Cluster và Memcached với Twemproxy cung cấp khả năng mở rộng tự động, phân phối dữ liệu và quản lý lỗi.



#### Redis Cluster:

Redis Cluster cung cấp giải pháp sharding tự động với khả năng phát hiện lỗi và tự động chuyển đổi dự phòng. Dữ liệu được phân phối qua nhiều node và tổ chức thành các slot (16,384 hash slots).

1	+-----+	+-----+	+-----+
2	Node1	Node2	Node3
3	Slots	Slots	Slots
4	0-5460	5461-	10923-
5		10922	16383
6	+-----+	+-----+	+-----+

Mỗi key được gán cho một slot cụ thể sử dụng hàm  $\text{CRC16}(\text{key}) \bmod 16384$ , và mỗi node quản lý một tập hợp các slot.

#### Memcached với Proxy:

Twemproxy hoặc mcrouter được sử dụng như một proxy trước nhiều instance Memcached, xử lý sharding và cân bằng tải:

1	+-----+
2	Twemproxy
3	+-----+
4	/   \
5	/   \

6	+-----+ +-----+ +-----+
7	MC1     MC2     MC3
8	+-----+ +-----+ +-----+

#### 5.6.2.4. Cache-Aside Pattern trong Microservice

Mẫu Cache-Aside là chiến lược phổ biến trong môi trường microservice, với mỗi service quản lý cache của riêng mình:

1	Client → Service A → Cache A → Database
2	Client → Service B → Cache B → Database

- **Ưu điểm:**

- Mỗi service có quyền kiểm soát hoàn toàn đối với cache của mình
- Giảm phụ thuộc giữa các service
- Mỗi service có thể tối ưu chiến lược cache phù hợp với nhu cầu riêng

- **Thách thức:**

- Có thể dẫn đến dữ liệu trùng lặp giữa các service
- Khó đảm bảo tính nhất quán khi nhiều service cập nhật cùng một dữ liệu
- Quản lý phức tạp hơn với nhiều cache riêng biệt

#### 5.6.2.5. Geo-distributed Caching

Đối với ứng dụng toàn cầu, cache có thể được phân phối trên nhiều khu vực địa lý để giảm độ trễ cho người dùng ở các vị trí khác nhau.

- **Chiến lược:**

- **Local-first approach:** Ưu tiên phục vụ từ cache cục bộ trong khu vực của người dùng
- **Cross-region replication:** Sao chép dữ liệu quan trọng giữa các khu vực
- **Hệ thống phân cấp:** Sử dụng cache cục bộ kết hợp với cache toàn cầu

### 5.6.3. Tối Ưu Hiệu Suất Cache

#### 5.6.3.1. Eviction Policies (Chính sách loại bỏ)

- Lựa chọn chính sách loại bỏ phù hợp đóng vai trò quan trọng trong hiệu suất cache:

- **LRU (Least Recently Used):** Loại bỏ mục được truy cập lâu nhất, tối ưu cho dữ liệu truy cập theo thời gian
- **LFU (Least Frequently Used):** Loại bỏ mục ít được sử dụng nhất, tốt cho dữ liệu có mẫu truy cập ổn định
- **TTL (Time-To-Live):** Loại bỏ mục sau một khoảng thời gian nhất định, phù hợp với dữ liệu cần tính nhất quán cao

#### 5.6.3.2. Consistency Strategies (Chiến lược nhất quán)

- Trong hệ thống phân tán, tính nhất quán của cache là thách thức lớn:
  - **Write-through**: Cập nhật cache và database cùng lúc
  - **Write-behind/Write-back**: Cache được cập nhật ngay lập tức, database được cập nhật sau
  - **Cache invalidation**: Xóa mục khỏi cache khi dữ liệu thay đổi, buộc phải tải lại từ nguồn
  - **Versioning**: Gán phiên bản cho mục cache để phát hiện và xử lý sự không nhất quán

#### 5.6.3.3. Monitoring và Analytics

Giám sát cache trong môi trường microservice rất quan trọng:

- **Hit ratio**: Tỷ lệ yêu cầu được phục vụ từ cache
- **Eviction rate**: Tốc độ các mục bị loại bỏ
- **Memory usage**: Sử dụng bộ nhớ trên mỗi node
- **Response time**: Thời gian phản hồi cho các thao tác cache
- **Replication lag**: Độ trễ giữa master và slave

### 5.6.4. Các Công Nghệ Cache Phổ Biến cho Microservice

#### 5.6.4.1. Redis

Redis là giải pháp cache phổ biến nhất cho microservice nhờ tính linh hoạt, hiệu suất cao và các tính năng phong phú:

- **Redis Cluster**: Hỗ trợ sharding, replication và failover tự động
- **Redis Sentinel**: Giám sát và tự động failover cho cấu hình master-slave
- **Redis Enterprise**: Cung cấp giải pháp geo-distribution và quản lý tiên tiến

#### 5.6.4.2. Memcached

Memcached là giải pháp đơn giản nhưng mạnh mẽ:

- Hiệu suất cao cho các thao tác đơn giản
- Mở rộng dễ dàng qua client-side sharding
- Hỗ trợ tốt cho mô hình cache phân tán

#### 5.6.4.3. Hazelcast

Hazelcast là giải pháp in-memory data grid:

- Phân phối và xử lý dữ liệu song song
- Hỗ trợ nhiều cấu trúc dữ liệu phân tán
- Tích hợp tốt với các framework microservice như Spring Boot

#### 5.6.4.4. Couchbase

Couchbase kết hợp cache in-memory với database NoSQL:

- Kiến trúc memory-first cho hiệu suất cao
- Hỗ trợ replication và sharding tích hợp
- Cross Data Center Replication (XDCR) cho phân phối địa lý

#### 5.6.5. Các Ví Dụ Thực Tế

##### 5.6.5.1. Ví dụ 1: Redis Cluster với Sharding

Cấu hình Redis Cluster với 6 node (3 master và 3 replica):

```
1 # Khởi tạo các node master
2 redis-cli --cluster create 192.168.1.101:7000 192.168.1.102:7000
3 192.168.1.103:7000 \
4 --cluster-replicas 1 -a password
5 # Kết quả: hệ thống tự động thêm các replica
6 # M: Master, S: Slave/Replica
7 # M[0] → S[0]
8 # M[1] → S[1]
9 # M[2] → S[2]
```

Trong ứng dụng Java với Spring Data Redis:

```
1 @Configuration
2 public class RedisConfig {
3     @Bean
4     public RedisConnectionFactory connectionFactory() {
5         LettuceClientConfiguration clientConfig =
6             LettuceClientConfiguration.builder()
7                 .readFrom(ReadFrom.REPLICA_PREFERRED) // Ưu tiên đọc từ
8                 replica
9                 .build();
10
11         RedisClusterConfiguration clusterConfig = new
12             RedisClusterConfiguration();
13         clusterConfig.setClusterNodes(Arrays.asList(
14             new RedisNode("192.168.1.101", 7000),
15             new RedisNode("192.168.1.102", 7000),
16             new RedisNode("192.168.1.103", 7000),
17             // Thêm các replica
18             new RedisNode("192.168.1.104", 7000),
19             new RedisNode("192.168.1.105", 7000),
```



```

17         new RedisNode("192.168.1.106", 7000)
18     });
19
20     return new LettuceConnectionFactory(clusterConfig,
21         clientConfig);
22 }

```

### 5.6.5.2. Ví dụ 2: Consistent Hashing với Memcached

Sử dụng thư viện client Memcached với consistent hashing để phân phối dữ liệu đồng đều:

```

1  import net.spy.memcached.MemcachedClient;
2  import net.spy.memcached.KetamaConnectionFactory;
3
4  // Thiết lập kết nối với consistent hashing
5  KetamaConnectionFactory connectionFactory = new
6  KetamaConnectionFactory();
7  MemcachedClient client = new MemcachedClient(connectionFactory,
8      AddrUtil.getAddresses("server1:11211 server2:11211
9      server3:11211"));
10
11 // Lưu và truy xuất dữ liệu
12 client.set("key", 3600, value);
13
14 Object cachedValue = client.get("key");

```

## 5.6.6. Thách Thức và Giải Pháp

### 5.6.6.1. Cache Invalidation

- **Thách thức:** Khi dữ liệu thay đổi, cache cần được cập nhật hoặc làm mất hiệu lực để tránh phục vụ dữ liệu cũ.
- **Giải pháp:**
  - Sử dụng message broker như Kafka hoặc RabbitMQ để phát thông báo về thay đổi dữ liệu
  - Thiết lập TTL phù hợp cho dữ liệu
  - Triển khai cơ chế phát hiện thay đổi dữ liệu và cập nhật cache

### 5.6.6.2. Cache Stampede/Thundering Herd

- **Thách thức:** Nhiều yêu cầu đồng thời cho dữ liệu không có trong cache dẫn đến tải cao bất thường cho database.
- **Giải pháp:**

- Sử dụng khóa phân tán để giới hạn chỉ một service cập nhật cache
- Triển khai “cache priming” để tải trước dữ liệu quan trọng
- Sử dụng cơ chế “stale-while-revalidate” để phục vụ dữ liệu cũ trong khi cập nhật ngầm

#### **5.6.6.3. Quản Lý Bộ Nhớ**

- **Thách thức:** Cache tiêu thụ bộ nhớ đáng kể, cần quản lý hiệu quả.
- **Giải pháp:**
  - Thiết lập giới hạn bộ nhớ phù hợp cho mỗi node cache
  - Chọn chính sách loại bỏ tối ưu
  - Giám sát và cảnh báo khi sử dụng bộ nhớ vượt ngưỡng

#### **5.6.6.4. Nhất Quán Dữ Liệu trong Môi Trường Phân Tán**

- **Thách thức:** Đảm bảo tính nhất quán khi dữ liệu được phân phối qua nhiều node cache.
- **Giải pháp:**
  - Sử dụng phương pháp “eventual consistency” khi có thể
  - Triển khai cơ chế xác nhận và kiểm tra tính nhất quán
  - Sử dụng conflict resolution khi phát hiện không nhất quán