



**An-Najah National University
Faculty of Engineering and IT**

جامعة النجاح الوطنية
كلية الهندسة وتكنولوجيا المعلومات

Distributed Operating Systems

10636456

1st Semester | 2025-2026

Bazar-com Project | Part 2

Jana Moe'n Tahayni

12220650

Cache Consistency:

```
87 // search by id
88 app.get('/info/:id', async (req, res) => {
89   try {
90     const id = req.params.id;
91     console.log(` INFO Frontend: Info request for book ID: ${id}`);
92 
93     // serve from cache for read requests (if enabled)
94     const key = infoCacheKey(id);
95     if (CACHE_ENABLED && cache.has(key)) {
96       console.log(` ↗ Frontend: Cache HIT for book ID: ${id}`);
97       return res.json(cache.get(key));
98     }
99 
100    console.log(` 🚧 Frontend: Cache MISS for book ID: ${id}`);
101 
102    // pick a catalog replica (round-robin) for load balancing
103    const catalogReplicas = parseReplicas(CATALOG_REPLICAS, CATALOG);
104    const chosenCatalog = pickReplica(catalogReplicas, 'catalog');
105 
106    const response = await axios.get(`${chosenCatalog}/info/${id}`);
107 
108    console.log(` ✓ Frontend: Info retrieved for: "${response.data.title}"`);
109 
110    // store in cache (only for read requests)
111    if (CACHE_ENABLED) {
112      cache.set(key, response.data);
113      enforceCacheLimit();
114      console.log(` 💡 Frontend: Cached info for book ID: ${id} (cache size: ${cache.size})`);
115    }
}
```

As the script code from frontend service above explains when searching an information for a specific id book, the service searches first if the item is cached or not. If not, it communicates with catalog service to get the required book. Then, it stores the book in its cache for next time.

To answer the following questions:

Compute the average response time (query/buy) of your new systems. What is the response time with and without caching? How much does caching help?

For query part:

I will calculate average time for searching 10 values without enabling the cache, and 10 search values with enabling it.

For purchase part:

I will calculate average time for purchasing 5 times without enabling the cache, and another 5 times with enabling the cache.

Query Part:

Without cache:

```
JS app.js M docker-compose.yml M X
docker > docker-compose.yml
1 version: "3.9"
2
3 services:
4     # Frontend (NOT replicated)
5     fronted:
6         build:
7             context: ../fronted
8             dockerfile: Dockerfile
9         ports:
10            - "3000:3000"
11         environment:
12            - PORT=3000
13            - CATALOG_REPLICAS=http://catalog1:4001,http://catalog2:4002
14            - ORDER_REPLICAS=http://order1:5001,http://order2:5002
15            - CACHE_ENABLED=false
16            - CACHE_MAX=100
17         depends_on:
18            - catalog1
19            - catalog2
20            - order1
21            - order2
```

Example to one output:

The screenshot shows the Postman application interface. At the top, it says "My Collection / Get data". Below that, there's a "GET" method dropdown and a URL input field containing "http://localhost:3000/info/7". To the right of the URL is a "Send" button. Underneath the URL, there are tabs for "Docs", "Params", "Auth", "Headers (8)", "Body", "Scripts", "Settings", and "Cookies". The "Body" tab is selected and has a "raw" dropdown set to "JSON". The JSON response is displayed below, showing a single book object:

```
{}
1 {
2     "id": 7,
3     "title": "Spring in the Pioneer Valley",
4     "topic": "undergraduate school",
5     "price": 30,
6     "quantity": 5
7 }
```

```

fronted-1 | Frontend: Info request for book ID: 7
catalog1-1 | Info request for book ID: 7 - "Spring in the Pioneer Valley"
fronted-1 | Frontend: Cache MISS for book ID: 7
fronted-1 | Frontend: Info retrieved for: "Spring in the Pioneer Valley"

```

Average time for 10 searches by id operations: 63.5ms

With cache enable:

```

3  services:
4    # Frontend (NOT replicated)
5    fronted:
6      build:
7        context: ../fronted
8        dockerfile: Dockerfile
9      ports:
10     - "3000:3000"
11    environment:
12    - PORT=3000
13    - CATALOG_REPLICAS=http://catalog1:4001,http://catalog2:4002
14    - ORDER_REPLICAS=http://order1:5001,http://order2:5002
15    - CACHE_ENABLED=true
16    - CACHE_MAX=100
17  depends_on:

```

Output example:

```

fronted-1 | Frontend: Info request for book ID: 5
fronted-1 | Frontend: Cache HIT for book ID: 5

```

The screenshot shows a browser-based API testing interface. At the top, there is a header bar with buttons for 'GET', 'Send', and other options. Below the header, there are tabs for 'Docs', 'Params', 'Auth', 'Headers (8)', 'Body' (which is currently selected), 'Scripts', 'Settings', 'Cookies', 'Schema', and 'Beautify'. The 'Body' tab shows a raw JSON response. The response status is '200 OK' with a timestamp of '15 ms' and a size of '341 B'. The response body is a JSON object representing a book:

```

{
  "id": 5,
  "title": "How to finish Project 3 on time",
  "topic": "undergraduate school",
  "price": 30,
  "quantity": 5
}

```

Average time for 10 searches by id operations: 20.7ms

Performance for caching in search operation: $(63.5 - 20.7)/63.5 *100\% = 67.4\%$

For Purchase Part:

With enabled cache:

```
fronted-1 | Frontend: Purchase request for book ID: 5
catalog1-1 | Info request for book ID: 5 - "How to finish Project 3 on time"
order1-1 | Purchase request for: "How to finish Project 3 on time" - Current quantity: 5
fronted-1 | Cache invalidate for book ID: 5 - existed: true

catalog1-1 | Sent cache invalidate to frontend for book ID: 5
fronted-1 | Cache invalidate for book ID: 5 - existed: false
catalog2-1 | Sent cache invalidate to frontend for book ID: 5
catalog2-1 | Updated book ID: 5 - Quantity: 4, Price: 30
catalog1-1 | Replicated update of book ID 5 to peer replica
catalog1-1 | Updated book ID: 5 - Quantity: 4, Price: 30
order2-1 | Received replicated order ID 1 and saved locally
order1-1 | Replicated order ID 1 to peer replica

fronted-1 | Frontend: Purchase successful - Order ID: 1
order1-1 | Purchase successful! Order ID: 1, Remaining quantity: 4
```

POST <http://localhost:3000/purchase/6> Send

Docs Params Auth Headers (8) Body Scripts Settings Cookies

raw JSON Schema Beautify

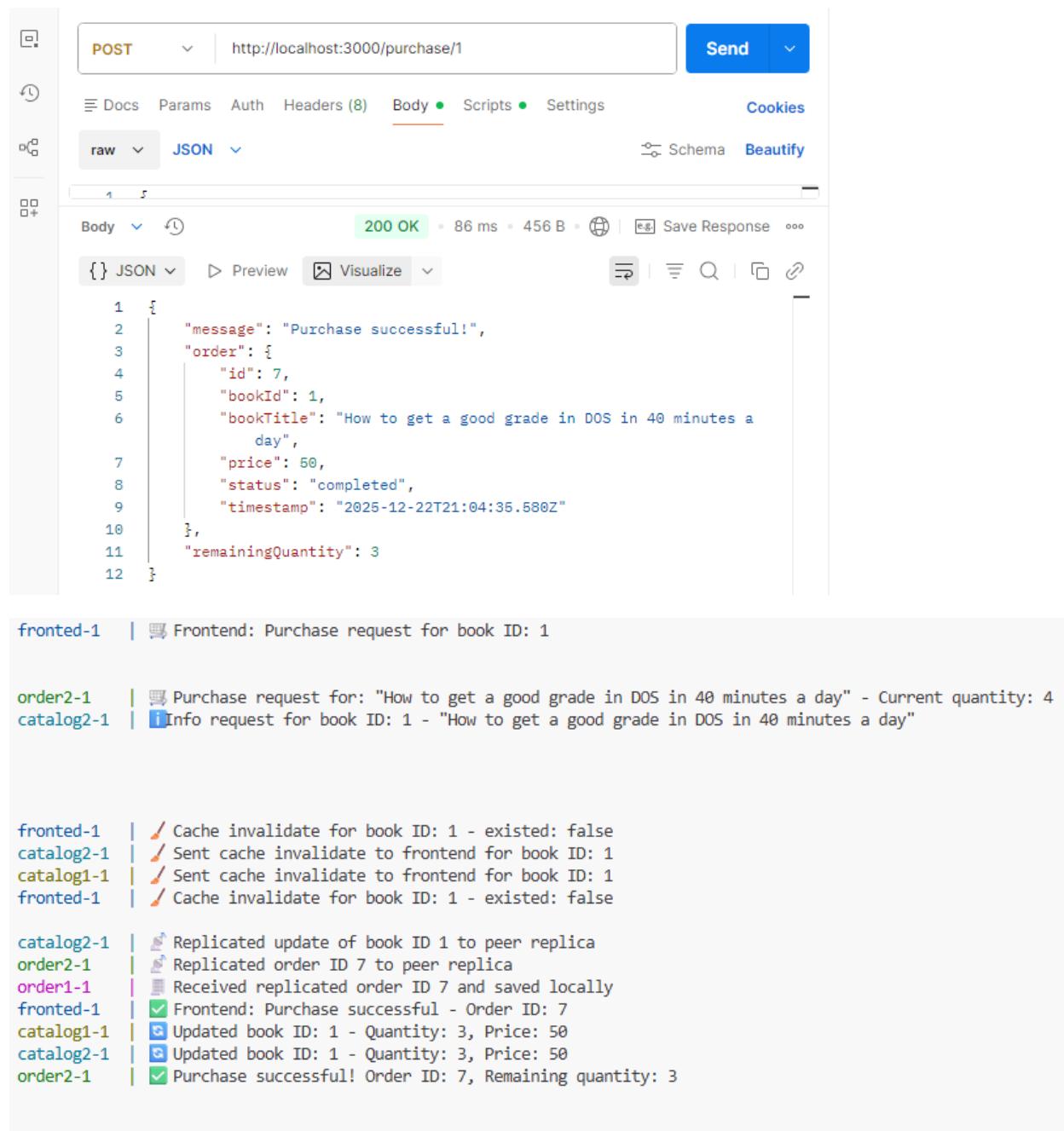
Body 200 OK 79 ms 437 B Save Response

{ } JSON Preview Visualize

```
1 "message": "Purchase successful!",  
2 "order": {  
3     "id": 6,  
4     "bookId": 6,  
5     "bookTitle": "Why theory classes are so hard.",  
6     "price": 30,  
7     "status": "completed",  
8     "timestamp": "2025-12-22T21:00:09.376Z"  
9 },  
10     "remainingQuantity": 4  
11 },  
12 }
```

Average: 148.2 ms

With disabled cache:



POST http://localhost:3000/purchase/1 Send

Docs Params Auth Headers (8) Body Scripts Settings Cookies

raw JSON Schema Beautify

Body Preview Visualize

200 OK 86 ms 456 B Save Response

```
{ "message": "Purchase successful!", "order": { "id": 7, "bookId": 1, "bookTitle": "How to get a good grade in DOS in 40 minutes a day", "price": 50, "status": "completed", "timestamp": "2025-12-22T21:04:35.580Z" }, "remainingQuantity": 3 }
```

fronted-1 | Frontend: Purchase request for book ID: 1

order2-1 | Purchase request for: "How to get a good grade in DOS in 40 minutes a day" - Current quantity: 4

catalog2-1 | Info request for book ID: 1 - "How to get a good grade in DOS in 40 minutes a day"

fronted-1 | Cache invalidate for book ID: 1 - existed: false

catalog2-1 | Sent cache invalidate to frontend for book ID: 1

catalog1-1 | Sent cache invalidate to frontend for book ID: 1

fronted-1 | Cache invalidate for book ID: 1 - existed: false

catalog2-1 | Replicated update of book ID 1 to peer replica

order2-1 | Replicated order ID 7 to peer replica

order1-1 | Received replicated order ID 7 and saved locally

fronted-1 | Frontend: Purchase successful - Order ID: 7

catalog1-1 | Updated book ID: 1 - Quantity: 3, Price: 50

catalog2-1 | Updated book ID: 1 - Quantity: 3, Price: 50

order2-1 | Purchase successful! Order ID: 7, Remaining quantity: 3

Average: 74.2 ms

Operation	Avg. With cache	Avg. Without cache	Improvement
Query	20.7	63.5	67.4%

Purchase	148.2	74.2	0.27%
----------	-------	------	-------

The average response time for purchase operations with caching enabled was slightly higher than without caching. This behavior is expected, since purchase requests are writing operations that must always contact the order and catalog services.

Additionally, when caching is enabled, the catalog service sends a cache invalidation request to the frontend prior to updating the database, introducing a small overhead. As a result, caching significantly improves query latency but does not benefit and may slightly increase the latency of purchase operation.

To answer this question:

Construct a simple experiment that issues orders or catalog updates (i.e., database writes) to invalidate the cache and maintain cache consistency. What is the overhead of cache consistency operations? What is the latency of a subsequent request that sees a cache miss?

Experiment:

1. Search for book with id = 6 when cache is null:

The screenshot shows the Postman interface with a GET request to `http://localhost:3000/info/6`. The response is a `200 OK` with a duration of `963 ms` and a size of `341 B`. The JSON response body is:

```

1 {
2   "id": 6,
3   "title": "Why theory classes are so hard.",
4   "topic": "undergraduate school",
5   "price": 30,
6   "quantity": 5

```

Note: time is 963 ms.

Cache miss =>

```
fronted-1 | i Frontend: Info request for book ID: 6
fronted-1 | 🏫 Frontend: Cache MISS for book ID: 6
catalog1-1 | i Info request for book ID: 6 - "Why theory classes are so hard."
fronted-1 | ✓ Frontend: Info retrieved for: "Why theory classes are so hard."
fronted-1 | 🌐 Frontend: Cached info for book ID: 6 (cache size: 1)
```

2. Search same book again [now it is in cache]:

GET http://localhost:3000/info/6

Body (8) Body Scripts Settings Cookies

raw JSON Schema Beautify

Body 200 OK 38 ms 341 B Save Response

{ } JSON Preview Visualize

```
1 {
2   "id": 6,
3   "title": "Why theory classes are so hard.",
4   "topic": "undergraduate school",
5   "price": 30,
6   "quantity": 5
7 }
```

Note: time is 38 ms.

Cache hit => no going to catalog service

```
fronted-1 | i Frontend: Info request for book ID: 6
fronted-1 | ⚡ Frontend: Cache HIT for book ID: 6
```

3. Update operation [patch] on book with id = 6 to see the invalidation.

The screenshot shows the Postman interface. A PATCH request is made to `http://localhost:4001/update/6`. The request body is a JSON object with a single field `quantity` set to 8. The response status is 200 OK, with a response time of 974 ms and a response size of 375 B. The response body is a JSON object containing a message "Book updated" and a book object with ID 6, title "Why theory classes are so hard.", topic "undergraduate school", price 30, and quantity 8.

```
1 {  
2   "quantity": 8  
3 }  
4  
Body 200 OK 974 ms 375 B Save Response  
{ } JSON Preview Visualize  
1 {  
2   "message": "Book updated",  
3   "book": {  
4     "id": 6,  
5     "title": "Why theory classes are so hard.",  
6     "topic": "undergraduate school",  
7     "price": 30,  
8     "quantity": 8  
9   }  
10 }
```

```
catalog1-1 | ✓ Sent cache invalidate to frontend for book ID: 6  
fronted-1 | ✓ Cache invalidate for book ID: 6 - existed: true  
fronted-1 | ✓ Cache invalidate for book ID: 6 - existed: false  
catalog2-1 | ✓ Sent cache invalidate to frontend for book ID: 6  
catalog2-1 | ✅ Updated book ID: 6 - Quantity: 8, Price: 30  
catalog1-1 | 📥 Replicated update of book ID 6 to peer replica  
catalog1-1 | ✅ Updated book ID: 6 - Quantity: 8, Price: 30
```

4. Search for book with id = 6 to ensure it will be missed from cache.

The screenshot shows the Postman interface. A GET request is made to `http://localhost:3000/info/6`. The request body is a JSON object with a single field `quantity` set to 8. The response status is 200 OK, with a response time of 191 ms and a response size of 341 B. The response body is a JSON object containing a message and a book object with ID 6, title "Why theory classes are so hard.", topic "undergraduate school", price 30, and quantity 8.

```
1 {  
2   "quantity": 8  
3 }  
4  
Body 200 OK 191 ms 341 B Save Response  
{ } JSON Preview Visualize  
1 {  
2   "id": 6,  
3   "title": "Why theory classes are so hard.",  
4   "topic": "undergraduate school",  
5   "price": 30,  
6   "quantity": 8  
7 }
```

```
fronted-1 | i Frontend: Info request for book ID: 6
fronted-1 | 🏫 Frontend: Cache MISS for book ID: 6
catalog2-1 | i Info request for book ID: 6 - "Why theory classes are so hard."
fronted-1 | ✓ Frontend: Info retrieved for: "Why theory classes are so hard."
fronted-1 | 🌈 Frontend: Cached info for book ID: 6 (cache size: 1)
```

The first query following a write operation experienced a cache miss and therefore had higher latency compared to cache hits.

Load Balancing:

To distribute client requests across replicated services, we implemented load balancing at the frontend layer. The frontend maintains a list of available catalog and order replicas and routes incoming requests using a simple round-robin algorithm.

Each new request is forwarded to the next replica in the list, ensuring an even distribution of workload across replicas.

This approach improves scalability and fault tolerance, as the system can continue to serve requests even if one replica becomes overloaded.

We chose round-robin load balancing due to its simplicity and low overhead, which is appropriate for our small-scale replicated system.

```
20 // NEW: round-robin indices for load balancing
21 let catalogIndex = 0;
22 let orderIndex = 0;
23
24 // NEW: helper to parse replicas list, fallback to single URL
25 const parseReplicas = (listStr, fallbackUrl) => {
26   if (!listStr) return [fallbackUrl];
27   const list = listStr.split(',').map(s => s.trim()).filter(Boolean);
28   return list.length > 0 ? list : [fallbackUrl];
29 };
30
31 // NEW: choose a replica using round-robin
32 const pickReplica = (replicas, type) => {
33   if (type === 'catalog') {
34     const chosen = replicas[catalogIndex % replicas.length];
35     catalogIndex++;
36     return chosen;
37   }
38   const chosen = replicas[orderIndex % replicas.length];
39   orderIndex++;
40   return chosen;
41 };
42
```

```
order1-1 | ✅ Purchase successful! Order ID: 6, Remaining quantity: 4
fronted-1 | 🚧 Cache invalidate for book ID: 1 - existed: false
catalog1-1 | 📖 Updated book ID: 1 - Quantity: 4, Price: 50
catalog2-1 | 📖 Updated book ID: 1 - Quantity: 4, Price: 50

fronted-1 | ✅ Frontend: Purchase successful - Order ID: 6
fronted-1 | 📄 Frontend: Purchase request for book ID: 1

order2-1 | 📄 Purchase request for: "How to get a good grade in DOS in 40 minutes a day" - Current quantity: 4
catalog2-1 | 📖 Info request for book ID: 1 - "How to get a good grade in DOS in 40 minutes a day"
```

Docker-aization:

The system was fully containerized using Docker to simplify deployment and ensure consistency across environments.

Each service (frontend, catalog replicas, and order replicas) runs in a separate container with its own runtime and dependencies.

Docker Compose was used to orchestrate the multi-container setup, defining service dependencies, environment variables, exposed ports, and persistent volumes.

Each replica uses a dedicated volume to store its local database file, ensuring data persistence across container restarts.

```
docker > 🌐 docker-compose.yml
 1 version: "3.9"
 2
 3 services:
 4   # Frontend (NOT replicated)
 5   fronted:
 6     build:
 7       context: ../fronted
 8       dockerfile: Dockerfile
 9     ports:
10       - "3000:3000"
11     environment:
12       - PORT=3000
13       - CATALOG_REPLICAS=http://catalog1:4001,http://catalog2:4002
14       - ORDER_REPLICAS=http://order1:5001,http://order2:5002
15       - CACHE_ENABLED=true
16       - CACHE_MAX=30
17     depends_on:
18       - catalog1
19       - catalog2
20       - order1
21       - order2
22
23   # Catalog replicas
24   catalog1:
25     build:
26       context: ../catalog
27       dockerfile: Dockerfile
28     ports:
29       - "4001:4001"
30     environment:
31       - PORT=4001
32       - DB_FILE=/data/catalog.json
33       - PEER_URL=http://catalog2:4002
34       - FRONTEND_URL=http://fronted:3000
35     volumes:
36       - catalog1_data:/data
37
38   catalog2:
39     build:
40       context: ../catalog
41       dockerfile: Dockerfile
42     ports:
43       - "4002:4002"
44     environment:
45       - PORT=4002
46       - DB_FILE=/data/catalog.json
47       - PEER_URL=http://catalog1:4001
48       - FRONTEND_URL=http://fronted:3000
49     volumes:
50       - catalog2_data:/data
```

```
55 # Order replicas
56 order1:
57   build:
58     context: ../order
59     dockerfile: Dockerfile
60   ports:
61     - "5001:5001"
62   environment:
63     - PORT=5001
64     - DB_FILE=/data/orders.json
65     - PEER_URL=http://order2:5002
66     - CATALOG_REPLICAS=http://catalog1:4001,http://catalog2:4002
67     - CATALOG_URL=http://catalog1:4001
68     - FRONTEND_URL=http://fronted:3000
69   volumes:
70     - order1_data:/data
71   depends_on:
72     - catalog1
73     - catalog2
74
75 order2:
76   build:
77     context: ../order
78     dockerfile: Dockerfile
79   ports:
80     - "5002:5002"
81   environment:
82     - PORT=5002
83     - DB_FILE=/data/orders.json
```

How to run the project:

Inside VS code:

- Open terminal in docker folder
- Type: docker compose up -build