Miniazon Architecture

High-Performance Data Structures & Algorithms

An engineering deep-dive into hybrid storage systems.

Executive Summary: The Hybrid Engine

The Problem

Standard SQL databases are optimized for storage, not high-frequency interactive speed. Searching via `LIKE %query%` requires scanning every row ($O(N)$), which causes latency at scale.

The Solution

Miniazon implements a **Hybrid Architecture**:

- **Performance Layer:** Custom Data Structures in RAM for sub-millisecond operations.
- **Storage Layer:** SQLite for persistent data reliability.

1. The Search Engine: Inverted Index

Why not SQL?

Scanning a database row-by-row is slow. Instead, we use an **Inverted Index**.

Think of the index at the back of a textbook. Instead of reading every page to find a word, you look at the word to find the page numbers.

> **Concept:** Hash Table + Sets
>
> Maps a single keyword to a set of multiple Product IDs.

Abstract geometric diagram of an inverted index data structure

Implementation: search_index.py

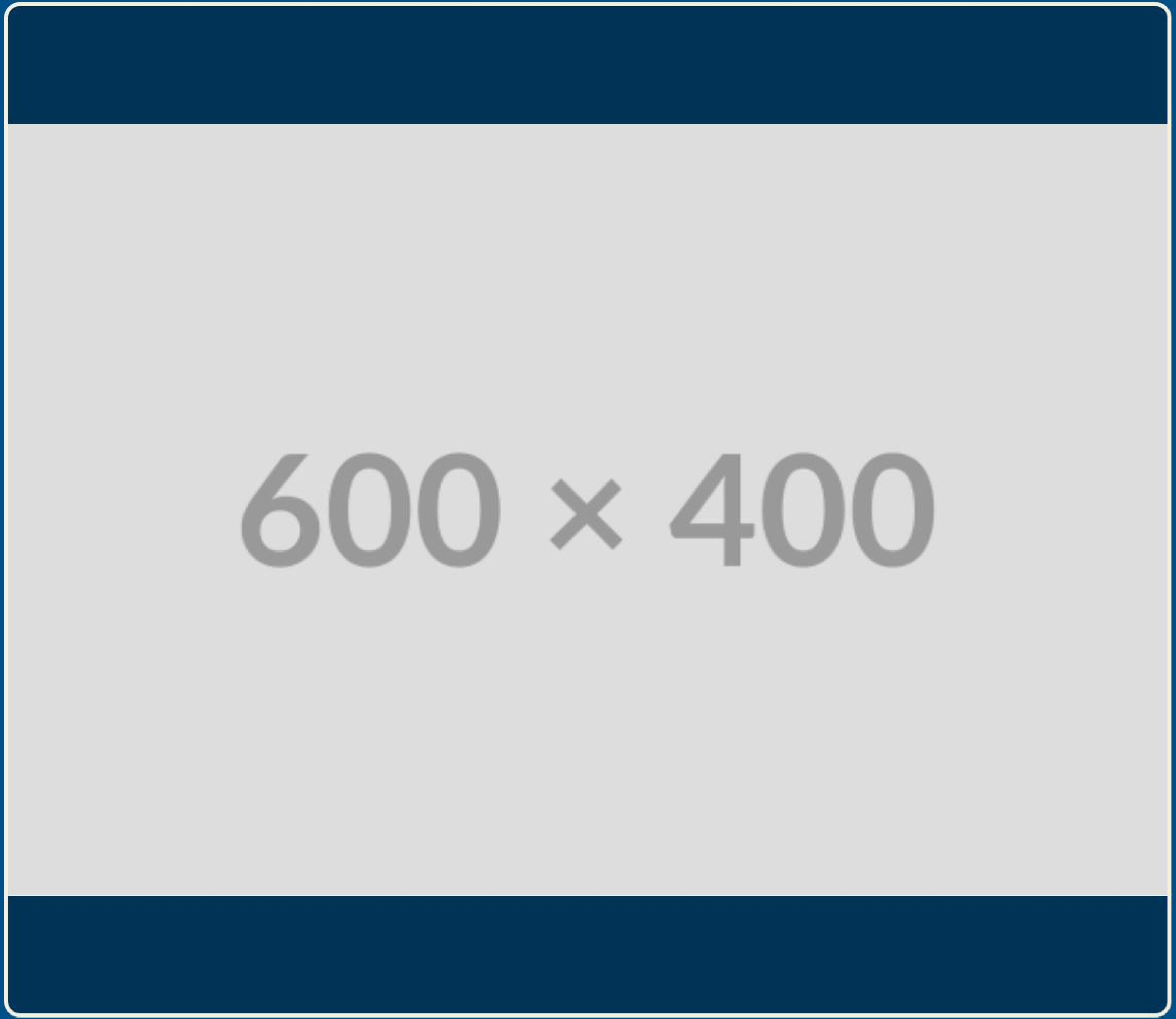
```
class SearchIndex:
    def __init__(self):
        # Hash table mapping words to product sets
        self.index = {}

    def search(self, query):
        # ... logic to split query ...
        # Intersect sets (AND logic)
        results &= self.index[word]
```

Complexity Analysis

- **Add Product:** $O(k)$
  Depends only on word count (k), not database size.
- **Search:** $O(m)$
  Depends only on query length (m).

"I avoided the Slow Scan ($O(N)$). I went straight to the answer."

2. The Shopping Cart: Nested Hash Table



The "Holy Grail" of Speed

If we used a simple list, updating a cart would require looping through items to find the right one. That is inefficient.

We use a **Nested Hash Table** (Dictionary of Dictionaries):

- **Outer Key:** User ID
- **Inner Key:** Product ID
- **Value:** Quantity

Implementation: shopping_cart.py

Constant Time Access

This structure allows us to access any specific item for any user instantly, without looping.

Complexity: $O(1)$

No matter if I have 10 users or 10 million users, adding an item takes exactly 1 step.

```python
self.carts = {
   "erosenpai_1337": { # Outer Key (User)
       1: 5            # Inner Key -> Value
   }
}

def add_item(self, user_id, prod_id, qty):
    # Direct access, no loops
    self.carts[user_id][prod_id] = qty
```

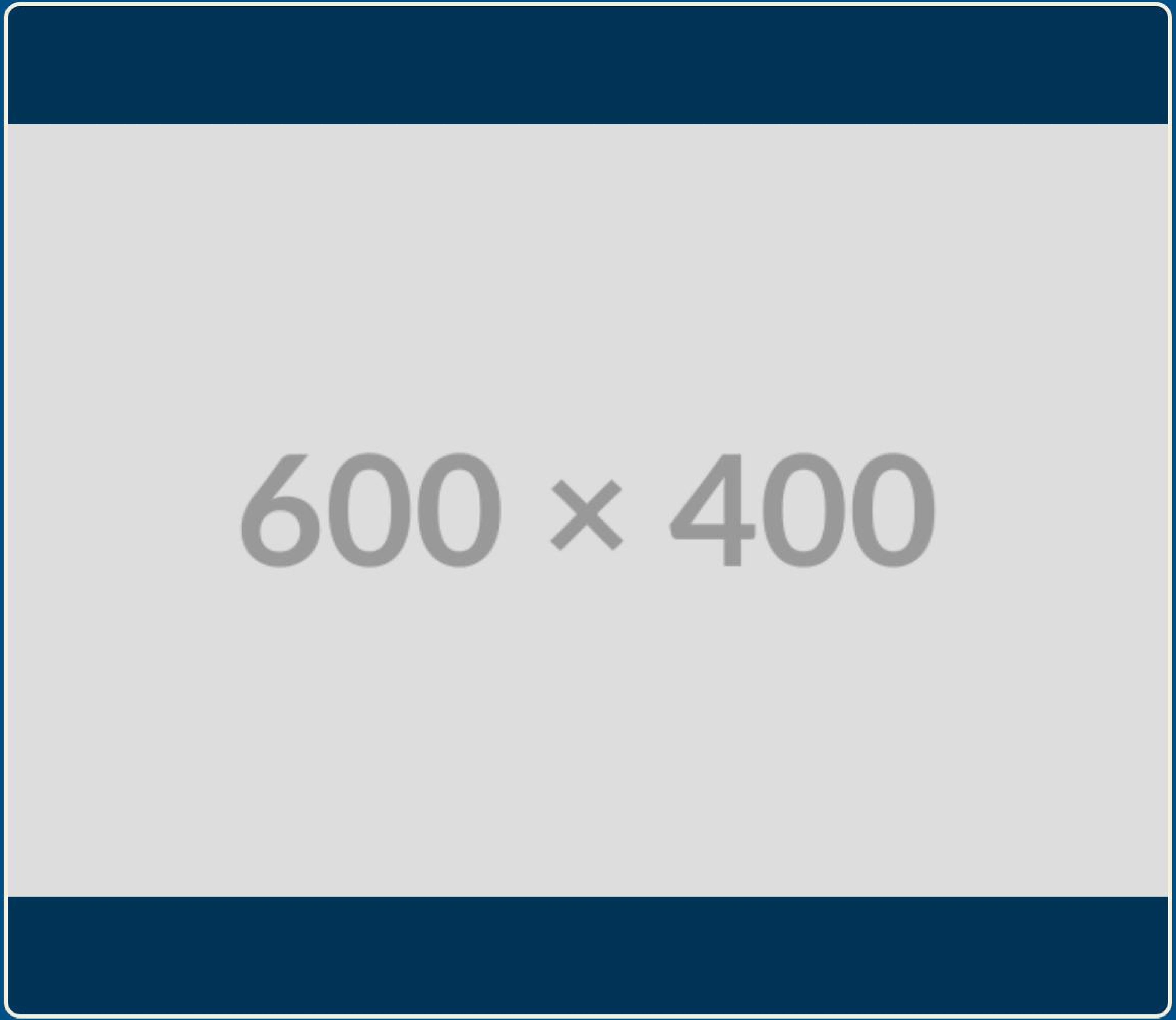
3. Recommendations: Product Graph

Modeling Relationships

How do we know that a "Laptop" is related to a "Mouse"? Storing this in a grid (Matrix) is wasteful because most products aren't related.

We use an **Adjacency List**. Each product holds a list of its "friends" (neighbors).

Type: Undirected Graph.



Implementation: recommendations.py

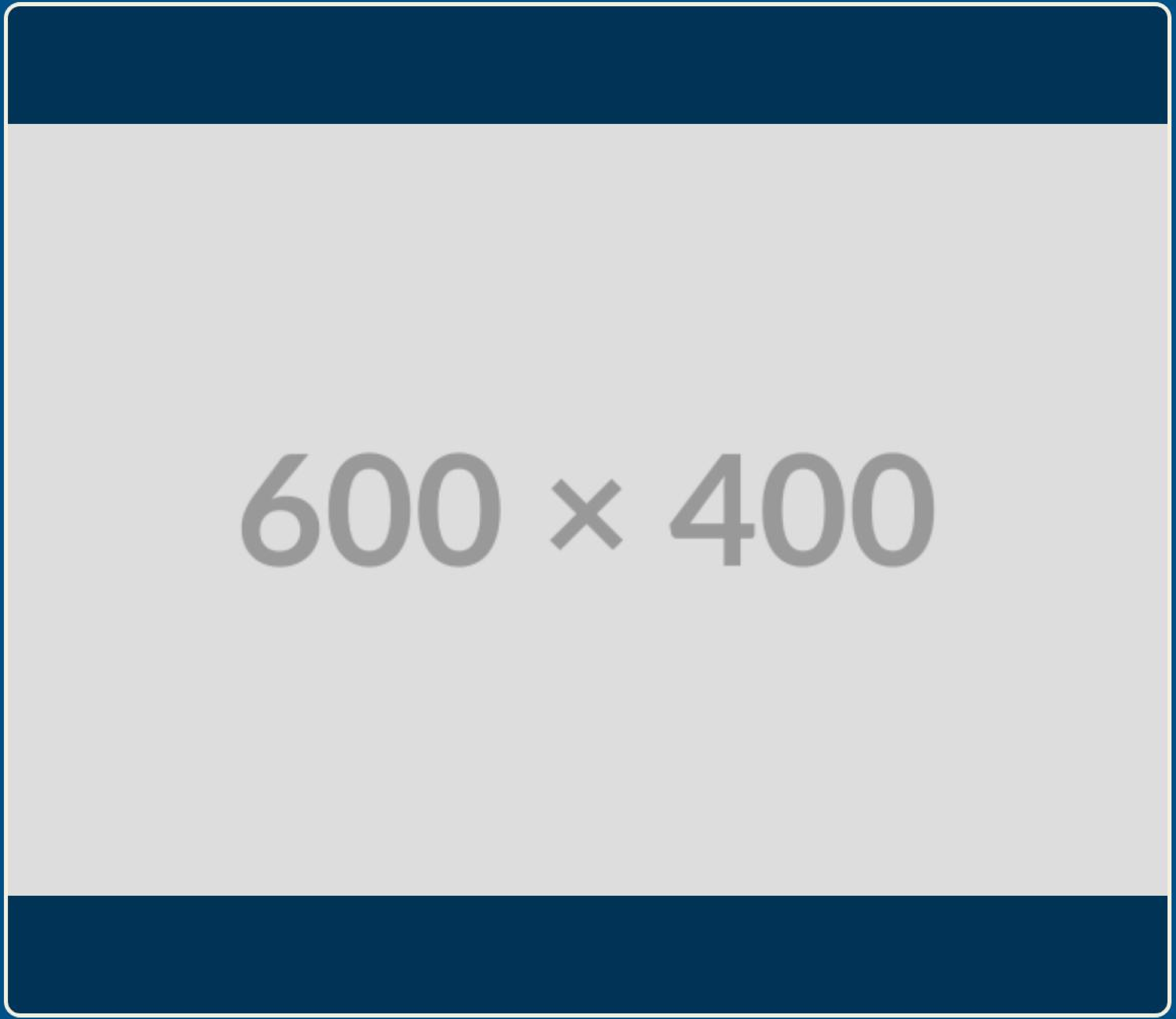
```
self.graph = {
   1: [2, 3, 4]  # Laptop -> [Mouse, Keyboard, Monitor]
}

def add_relationship(self, a, b):
    # Bidirectional logic
    self.graph[a].append(b)
    self.graph[b].append(a)
```

Efficient Traversal

Complexity: $O(1)$ (Effective)

Writing a name in an address book takes 1 step. Finding a product's friends is instantaneous because we just retrieve the list directly from the dictionary key.

System Architecture: RAM vs Disk



Search & Cart Flow

1. **User Action:** User types "Mouse" or adds item.

2. **RAM Processing:** Python uses the Inverted Index or Hash Map to find IDs or update Quantities. (Fast)

3. **SQL Storage:** Only accessed when we need rich details (Image, Price) for the final display. (Slow)

Recommendation Workflow

View Product

User visits "Laptop" page.

Graph Query

Python checks neighbor list in RAM.

ID Retrieval

Returns IDs: [Mouse, Monitor].

Render

Frontend displays "Also bought".

Algorithmic Efficiency

Comparison of Miniazon's engineered approach vs. a naive database approach.

Feature	DSA Implemented	Complexity	Naive Approach (SQL)
Search	Inverted Index	$O(k)$	$O(N)$ (Full Table Scan)
Cart	Nested Hash Map	$O(1)$	$O(N)$ (List Iteration)
Recs	Adjacency List	$O(1)$	$O(N^2)$ (Complex Joins)

Engineering, Not Just Coding.

Miniazon proves that intelligent data structure selection solves performance problems that raw database power cannot.

kwisdomk/Miniazon