Pagination Issue :

### **Problem in the Existing Model**

The current implementation faces significant challenges when querying CRFs for a large date range (e.g., more than 1 year, sometimes even slightly over 1 month). The key issues are:

1. **Out of Memory Error**: The backend fetches the entire dataset for the given date range and loads it into the frontend's RAM. If the data size exceeds the capacity of the browser's memory, it leads to an Out of Memory (OOM) error, crashing the page.
2. **Inefficient Pagination**: The current model does not implement efficient pagination. Instead of fetching paginated data from the backend, it loads the entire dataset first and then attempts pagination on the frontend.
3. **Scalability Issues**: As the data grows, the application becomes more prone to performance issues and crashes, making it non-scalable.
4. **High Resource Utilization**: The unnecessary loading of large datasets puts strain on the server and the frontend browser, affecting overall application performance.



### 

### 

### **Solutions:**

#### **Solution 1: Frontend-Side Pagination Using Browser Database (SQLite)**

In this approach, the data fetched from the backend is stored locally in the browser using a lightweight database such as **SQLite**. The pagination logic will operate on this local database instead of directly on the frontend's memory.

* **Workflow:**
  + Query data for the specified date range from the backend.
  + Store the data in the browser's SQLite database.
  + Implement pagination logic to fetch only the required data from SQLite and render it on the UI.
* **Advantages:**
  + No changes are required in the backend.
  + Reduces API calls to the backend as data is loaded once for a query.
  + Improves user experience by reducing load time and enabling smoother pagination.
* **Disadvantages:**
  + **Not Scalable**: If the data size increases significantly, even SQLite might fail to handle large datasets within browser memory.
  + Increased reliance on the frontend's capabilities, which vary across devices and browsers.
  + Large queries might still cause delays in the initial loading phase.



#### **Solution 2: Backend-Side Pagination with Redis Caching**

In this approach, the backend will implement server-side pagination and use **Redis** for caching to optimize performance.

* **Workflow:**
  + On receiving a date range query, check for data in the Redis cache.
    - If data is **present** in the cache (Cache Hit), return it directly to the frontend.
    - If data is **not present** (Cache Miss), fetch the required data from the database, store it in Redis for future use, and return paginated results to the frontend.
  + The frontend will request data page-by-page (e.g., 10-50 rows at a time), and the backend will ensure efficient pagination logic using limit-offset or cursor-based techniques.
* **Advantages:**
  + **Scalability**: Redis can handle large volumes of data efficiently, making this approach scalable.
  + **Improved Performance**: Caching reduces database calls, leading to faster responses.
  + **Memory Optimization**: Data is fetched and sent in chunks rather than loading everything into memory at once.
  + Reduces load on the frontend by implementing pagination at the backend level.
* **Disadvantages:**
  + Requires changes in the backend to implement Redis caching and pagination logic.
  + Redis introduces additional infrastructure, requiring setup and maintenance.



## 