Lecture V: The Item RAM Strikes Back!

Towards a practical dynamic bulk storage system for use in the real world

Instructor: Andrews54757

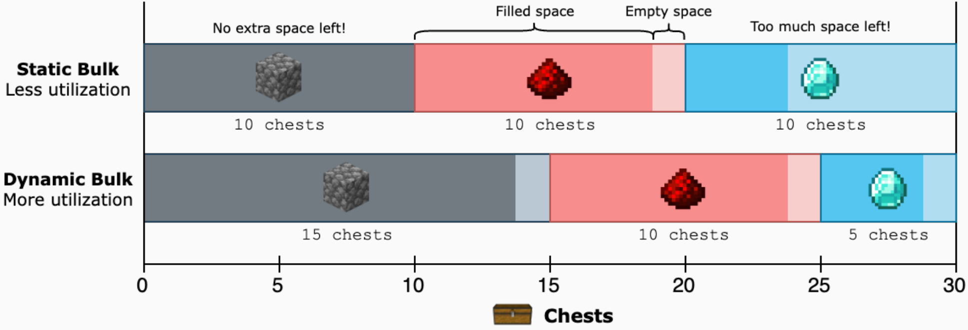
CSE269: Introduction to Encoded Storage
S∞ntech Annals

Overview

1. What is dynamic bulk?
2. Fast and compact data storage is hard
   - 2.1 Internal memory is too big, complex, and laggy
   - 2.2 Item RAMs are still too big
   - 2.3 Disk Drives are too slow
3. Solution: Linked-list based dynamic bulk
   - 3.1 Non-box item RAM is fast and compact
   - 3.2 Linked-list to the rescue
4. Conclusion and future steps

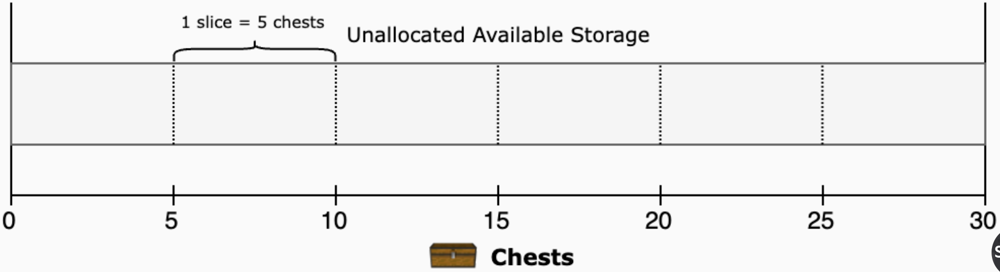
1. What is dynamic bulk?

- **Static bulk** has a fixed capacity for each item type
- **Dynamic bulk** automatically expands and shrinks capacity for items



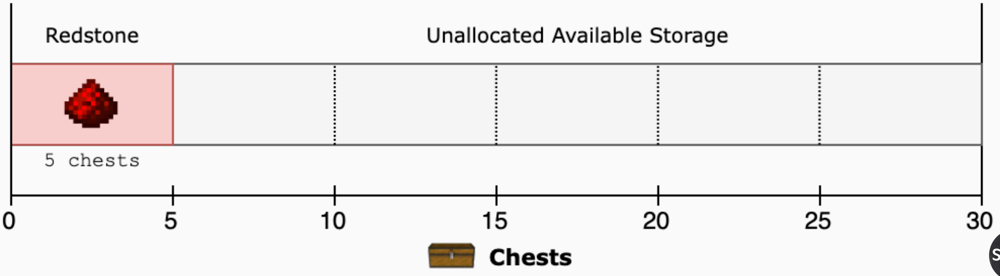
1. What is dynamic bulk? Contd.

- Divide your storage into **slices** of fixed size
- Each slice can store a fixed number of items
- The size of your slice determines the minimum non-zero amount of storage space you can allocate to an item type



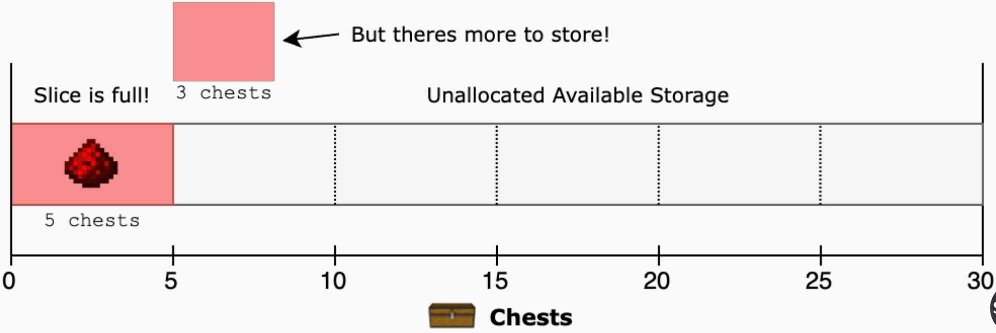
1. What is dynamic bulk? Contd.

- Let’s store 8 chests of redstone
  - Do we have a slice assigned to redstone? **No!**
  - Allocate an available slice to redstone
  - Now we have space to start storing redstone!



1. What is dynamic bulk? Contd.

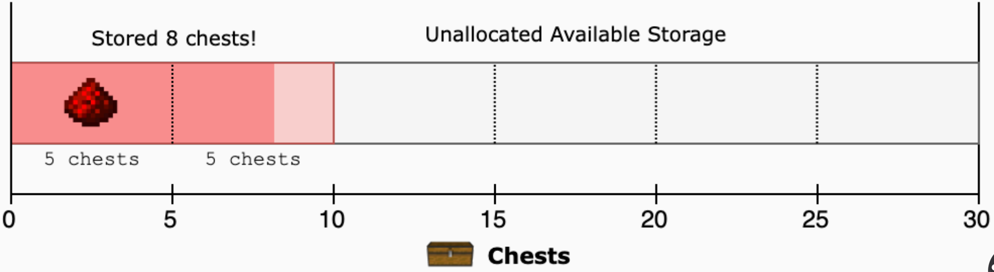
- Let’s store 8 chests of redstone
  - Fully filled one chest with 5 chests of redstone
  - But we have more 3 more chests of redstone to store!



Soon™ A5

1. What is dynamic bulk? Contd.

- Let's store 8 chests of redstone
  - One slice is **fully** filled with redstone (5 chests)
  - Another slice is **partially** filled with redstone (3 chests / 5 chests)

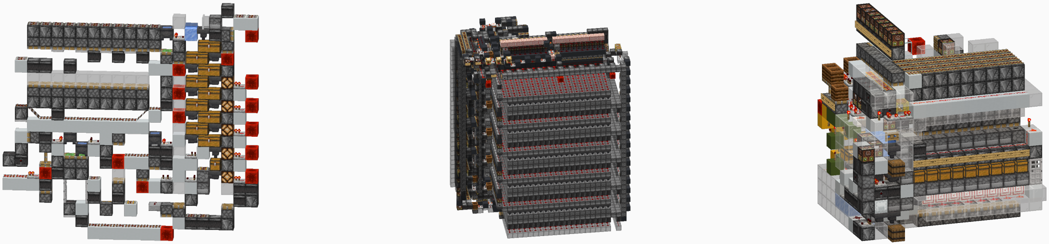


1. What is dynamic bulk? Contd.

- Retrieve 1 chest of redstone
  - Pull from the partially filled slice first!
  - Minimize internal fragmentation

2. Fast and compact data storage is hard

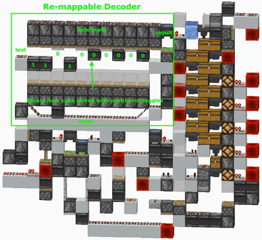
- We need to keep track of what slices are assigned to each item type
- Must store that information in a way that is quickly accessible
- Storage must be dense so that it is compact

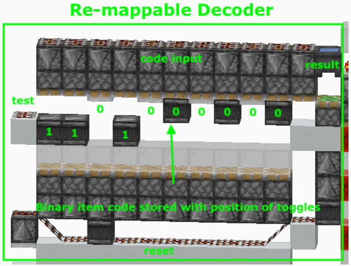


From left to right: PallaPalla's Dynamic Slice, 1000 Item RAM, Obi's Disk Drive

2.1 Internal memory is too big, complex, and laggy

- PallaPalla, original inventor of dynamic bulk, uses a binary re-mappable decoder to store assigned item code within each slice.
- Almost every piston in each slice's decoder is fired to write/test item codes.
- Allocation logic is internal to the slice, so each slice is large and complex.





2.1 Internal memory is too big, complex, and laggy. Contd.

- What if we externalized the slice allocation logic?
  - Each slice is now just a simple static encoded bulk unit
  - Store information about slice allocation in a separate memory unit
  - Externalized logic is simpler and more compact

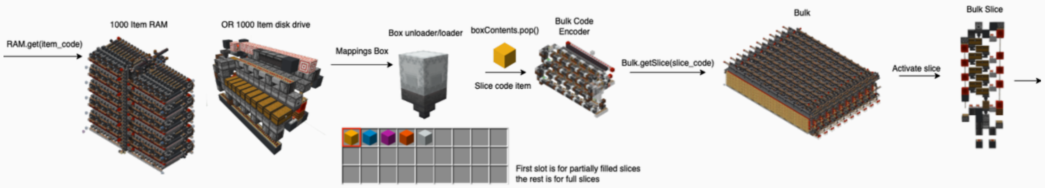


Figure 12: Diagram of dynamic bulk with externalized logic

2.1 Internal memory is too big, complex, and laggy. Contd.

- How is slice allocation information stored externally?
  - Represent slices with an encodeable item called the **slice code item**. One slice code item corresponds to one slice.
  - Maintain a list of slice code items in box dedicated to one item type
  - Store the list of slice code items in an external item memory unit in the address corresponding to the item type

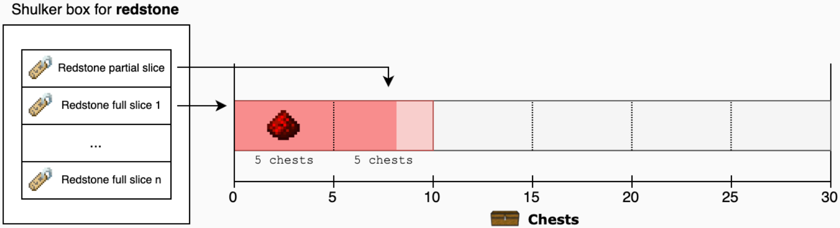


Figure 12: Externalized dynamic bulk data layout

2.2 Item RAMs are still too big

- An item RAM is a memory unit that stores items in an addressible way
- Each item type is assigned a unique address
- The item RAM can be read from and written quickly in constant time (O(1))

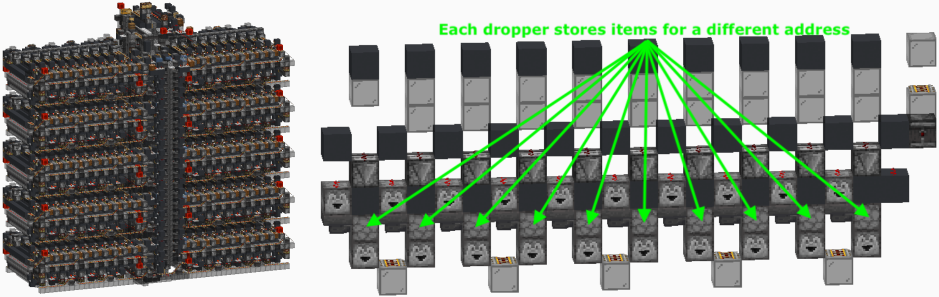


Figure 13-14: 1000 address item RAM and its slice. 46gt call latency.

2.2 Item RAMs are still too big. Contd.

- Unfortunately, item RAMs are very big.
- Latest advancements have doubled the density, but still too big.
- 17 x 41 x 33 blocks for 1000 addresses

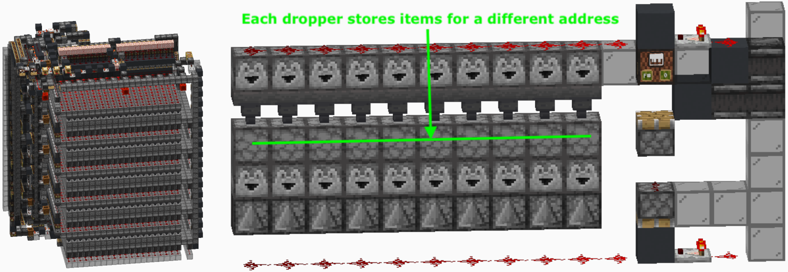


Figure 15-16: Improved 1000 address item RAM and its slice. 46gt call latency.

- What if, we stored multiple boxes from different addresses in a single block?
- Store up to 54 boxes in a double chest
- Cycle through the boxes to find the correct one

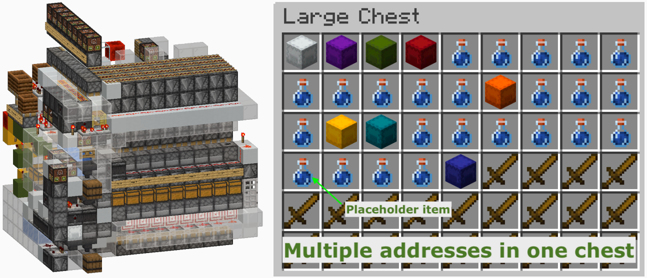


Figure 17-18: Obi's Disk Drive, 256 addresses.

2.3 Disk Drives are too slow. Contd.

- Cycling through slots is slow, 8gt per slot.
- Box at the 32nd slot takes a minimum 248gt to access, which is 12.4 seconds.
- Need something faster!

Figure 19: Minimum time in seconds to reach slot in a disk drive

3. Towards a better dynamic bulk

- How can we store the slice allocation information in a way that is fast and compact?
- Need a data storage that uses minimal space and can be accessed quickly
- What if we seperate the partial slice allocation information from the list of full slices?

3.1 Non-box item RAM is fast and compact

- If instead of storing boxes, we store a single non-box slice code item per address
- Can pack multiple addresses in a single shulker box for higher density

Figure 20: Non-box item RAM storage scheme

3.1 Non-box item RAM is fast and compact. Contd.

- We can quickly bring boxes to an ultrafast slot cycler to access the correct item
- Uses hoppercarts, which can pull items from shulker boxes every game tick
- 8x faster cycling than traditional disk drives. Item at the 20th slot takes only 1 second

Figure 21-22: 1 slot/gt slot cycler with internal layout

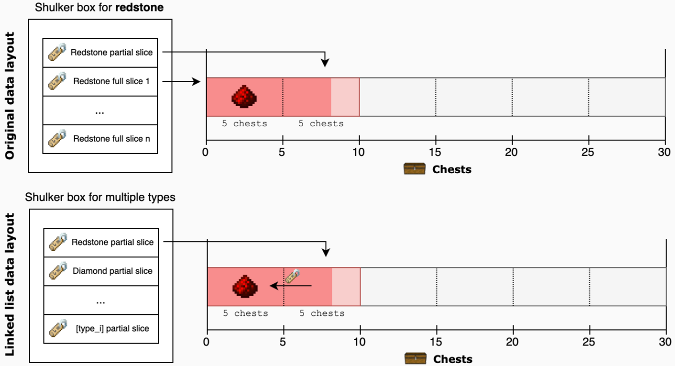
3.1 Non-box item RAM is fast and compact. Contd.

- Combine slot cycler with 50 item RAM (21gt call latency)
- Can access 1000 addresses within 55gt
- Result: Both fast and compact!

Figure 23: 1000 address non box item RAM

3.2 Linked-list to the rescue

- **Recall:** We need to be able to allocate more than one slice to an item type, but the non-box item RAM can only store one slice per address
- **Solution:** Store the next full slice's slice code item in the current slice to form a linked list



3.2 Linked-list to the rescue. Contd.

- Maintaining the linked list for **insertion** is simple.
  1. Obtain a slice code item at a slot corresponding to the item code using the 1000 non-box-item RAM.
  2. Insert boxes into the corresponding slice.
  3. If slice is full, pull a new slice code item from storage to allocate a new empty slice
  4. Put the now-full slice code item into the newly allocated slice and put the rest of the boxes into the newly allocated slice. Repeat 3-4 as needed.
  5. Store the remaining slice code item into the 1000 non-box-item RAM at the slot corresponding to the item code to finish insertion.

3.2 Linked-list to the rescue. Contd.

- Maintaining the linked list for **retreival** is also simple.
  1. Obtain a slice code item at a slot corresponding to the item code using the 1000 non-box-item RAM.
  2. Retrieve boxes from the corresponding slice. Test items if they are boxes, if one is not a box then hold it for later, it is the slice code item for the next slice.
  3. If slice is empty and you have found a code item for the next slice, then switch to the next slice. Put the now empty slice code item back into storage.
  4. Keep retreiving boxes, repeat 2-3 as needed.
  5. If you have found the slice code item for the next slice, put it back into the slice.
  6. Store the remaining slice code item into the 1000 non-box-item RAM at the slot corresponding to the item code to finish insertion.

4. Conclusion and future steps

- Linked-list based dynamic bulk has the potential to be both fast and compact
- Future work: Implement the linked-list based dynamic bulk and test its performance
- Further optimizations: Reduce the latency of the non-box item RAM
- **This is an assignment for you!** Unfortunately I have no time to implement this myself. Good luck!
    - I have posted the schematic of the 1000 address non-box item RAM in the description to help you get started