SKY MEMORY: USING A SATELLITE CONSTELLATION AS DISTRIBUTED MEMORY

INVENTORS:

THOMAS SANDHOLM

SAYAN MUKHERJEE

LIN CHENG

BERNARDO HUBERMAN

Background

Models based on the Transformer architecture have become the de facto standard for AI workloads. The versatility of the Transformer can be explained by an attention mechanism tracking correlations between tokens at different positions in a sequence. Originally conceived with text tokens for language translation, Large Language Models (LLMs) using the Transformer architecture are now adapted to solve almost any natural language processing (NLP) task, and have now been generalized to be Large Multimodal Models (LMMs) that process text and images, both still and video. We will discuss the text LLMs here, but the invention is applicable to any model employing the Transformer architecture, or any application that can benefit from edge caching.

An LLM processes a text prompt by first converting it to a sequence of tokens representing the semantics of the text. Next comes the prefilling phase, where this token sequence is passed through the attention layers of the LLM. The prefilling phase results in an output token sequence that is generated one token at a time (next token prediction). Converting the generated tokens to text again and controlling which token to produce given probability distributions over tokens is referred to the decoding phase. The whole process of generating an output text from a prompt text is referred to as inference.

With large prompts that are repetitive across generations, e.g. containing entire documents like in retrieval augmented generation (RAG) or long chat histories (contexts) the conversion of a token sequence to an output suffers large overheads.

In general, the attention layers cause the computational overhead to increase quadratically with the length of the context. As a result, early work to optimize this step cached the LLM intermediate outputs for common contexts in what is called key value caches (KVC), popularized in a seminal work called PagedAttention [1].

KVC stores key value caches for fixed ***blocks*** of context like how an operating system uses page caching. One important feature of this cache is that the sequence of blocks matters. So, a prompt is converted to an ordered sequence of blocks. When a new prompt appears it is enough to find a matching block in the cache that is furthest toward the end of the sequence of blocks, because that means that the entire preceding set of blocks to the matching block are also in the cache. Even for very small models (1B) the KVC for a small block (128 tokens) can be rather large (~3MB). The cache retrieval must be fast (from memory) so as not to defeat the purpose of speeding up the GPU computation. A trend to scale the inference to large models and contexts is to store the KVC across a memory hierarchy, such as GPU, CPU, network storage, cold disk storage and move the caches between the layers based on various last recently used (LRU) eviction

policies. For large contexts, the usual arguments for speed and computational complexity apply when comparing computation from scratch (quadratic complexity) versus retrieving from a cache, which is typically a lookup from a hash table (O(1) complexity). Thus, by trading memory storage for computation, caching can improve the latency and computational complexity of inference by an order of magnitude.

Recent advancement of low-earth-orbit (LEO) constellations for communication and earth observation has given rise to the concept of LEO edge, making use of the vast computational resources circling in orbit across the earth. Even full orbit-bound data centers have been proposed [2]. The advantage of a LEO constellation is that it is a highly distributed system with thousands of satellites connected with free-space optics inter satellite links (ISL) and always a just a hop a way from any point on earth. Highly directional phased array antennas have pushed down the latency to single digit milliseconds for ground communication. The key challenge is mobility where a single satellite may only be visible from a point on earth for 5-10 minutes.

The proposed invention expands the scope of cache memory to include the memory on LEO satellites. It thus increases cache hits which in turn improves speed of inference for LLMs, in particular the time to first token (TTFT) in the prefilling stage. This benefit applies to LLMs hosted both terrestrially and on satellites, and it is generalizable not just to key value caches for LLMs but to any use case with a cache distributed over multiple locations that needs to be accessed and set quickly.

Description

We propose a LEO satellite constellation hosted KVC that can fit seamlessly into current KVC memory hierarchies, to optimize LLM inference in general and prefilling in particular. Rather than hosting the KV Lists of an LLM during token generation, we find, pull in and deploy (in the GPU or other computing hardware when the inference computation takes place) the most appropriate full KVC of all KV Lists given a prompt, before running the inference computation on the prompt. We present a protocol for two use cases, LLMs hosted on earth and LLMs hosted on board satellites in a constellation. Finally, the invention is applicable to reducing the latency for any cache (not just a KVC for LLMs) that is distributed over different physical locations by exploiting the very low latencies of LEO links (see Table 1). Motivation A key-value cache can be stored in memory hierarchies, and the invention can be integrated into a stack of both faster and slower memory.

Table 1 is an approximate latency map of different memory types supported by the invention.

Table 1

Type	Latency
CPU	10-15 nanoseconds
GPU	50-100 nanoseconds
RDMA	2-5 microseconds
SSD	20-200 microseconds
HDD	2-20 milliseconds
NAS	30-40 milliseconds
LEO (current)	20-50 milliseconds
LEO (theoretical with laser)	2-4 milliseconds

We also note that from a single point on Earth, as many as 10-20 LEO satellites may be visible, which would allow direct communication in parallel with multiple satellites at a time.

The above numbers in Table 1 are for ground-to-satellite communication. In the on-board LLM case, for FSO in the LEO constellation ISL the latency is determined by inter-orbital and intra-orbital distances and speed of light. The distances can be computed with the following equations from [7]. Note that the within-orb latency does not change over time whereas the inter-orb latency is periodic.

$$D_N(t) = (r_E + h)\sqrt{2\left[1-\cos\left(\frac{2\pi}{N}\right)\right]\left[\cos^2\left(2\pi\frac{t}{T}+\frac{2\pi f}{M}\right)+\cos^2(i)\sin^2\left(2\pi\frac{t}{T}+\frac{2\pi f}{M}\right)\right]}$$

$$D_M = (r_E + h)\sqrt{2\left[1-\cos\left(\frac{2\pi}{M}\right)\right]}$$

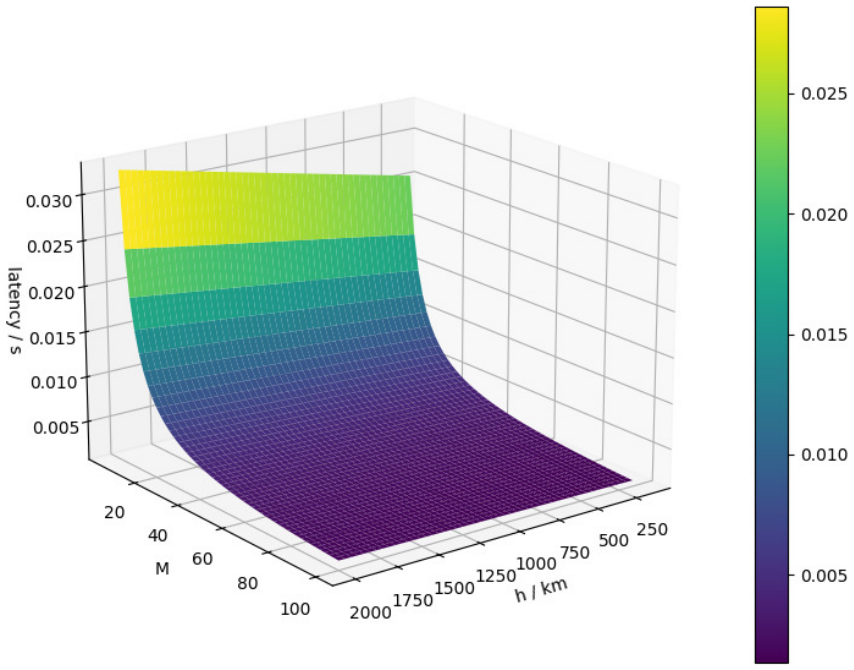
DN is the distance between neighbors in different orbital planes, DM the distance between neighbors in the same orbital plane, rE the radius of the earth, h the altitude of the satellites in the constellation, N the number of orbital planes in the constellation, M the number of satellites in an orbital plane in the constellation, T the orbital period, i the orbital inclination, f the Walker phase, and t the time variable.

This table from [7] gives examples of N and M for different constellations.

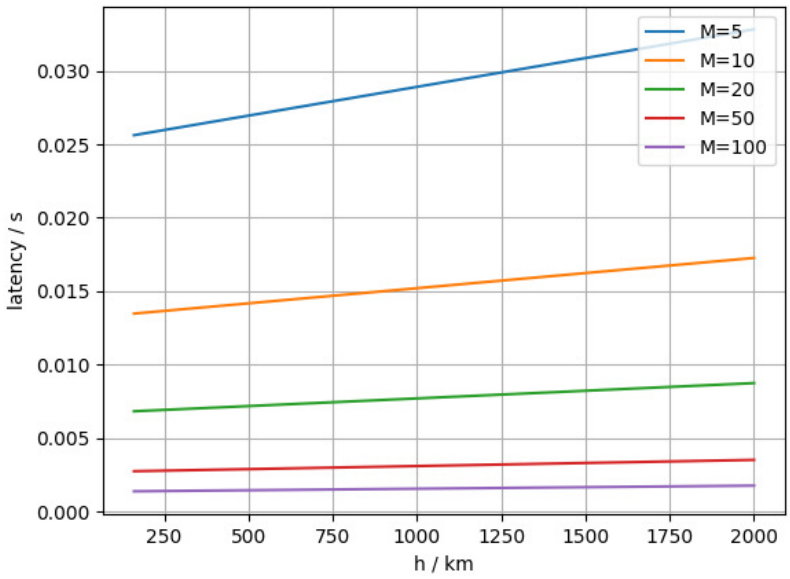
Table 2: table from [7] with examples of N and M for different constellations

LEO network	Constellation	N	M	Altitude (km)	Inclination (degrees)
Starlink	A	72	22	550	53.0
Starlink	B	5	75	1275	81.0
Kuiper	A	34	34	630	51.9
Kuiper	B	28	28	590	33.0

The figures below illustrate the intra-plane latency as a function of M and h.



(a)



(b)

Figure 1. Dependence of the intra-plane latency D_M on M and h: (a) three-dimensional plot; (b) contour plots of D_M vs h for different fixed values of M.

Extrapolating the numbers from these simulations, we can see that we get roughly the a latency between SSD and HDD (See Table 1) with about 50+ satellites in a plane or 50+ planes (<2 milliseconds).

Basic Protocol

Prompts are split into token blocks of a fixed size (e.g. 128 tokens). We start by hashing the first token block, then for the second token block, we hash that token block together with the hash of the first token block. Thus the hash for the second token block is actually the hash of the first two token blocks. Similarly, we hash the third token block together with the hash for the second token block (which is actually the hash of the first two token blocks), and so on, such that the hash for a token block in the cache is actually a hash of all the token blocks up to and including that token block. When finding a matching KVC it is thus enough to find the matching hash that is furthest to the end of the hash list. The lookup input is an ordered list of hashes representing the input prompt.

Given that these blocks can be large (several MB or GB even for modestly sized LLMs) we further split up each block into chunk s of fixed byte size (e.g. 6k bytes). Mapping from a chunk to a target server and how to look up a chunk can be done differently depending on the use case, discussed below. However it is worth noting that a failed lookup of a single chunk is enough to determine that the KVC does not exist for the queried block.

A baseline implementation of the protocol just computes the server (virtual chunk destination that will be mapped to a physical destination such as a satellite) to store on as chunk_id mod n where n is the max index of the server to host the chunk. Note that this allows for parallelism both in setting and getting a single KVC.

Each KVC cache entry is hence identified by the tuple (block_hash, chunk_id).

The basic protocol was inspired by [9].

LEO Constellation Model

A LEO constellation comprises a set of satellites orbiting earth in near circular paths. A constellation is identified by an altitude and an inclination angle shared across all its orbital paths. Each orbital path hosts the same number of equidistant satellites in a grid formation. There is wraparound both within a plane (first and last satellite can communicate) and across planes (first and last planes can

communicate) to form a 2D-Torus Mesh often referred to as +GRID. Even though ISL communication is performed via free space optics, geography matters and the closer the satellite is that hosts the cache to the one requesting it the better and more reliable the latency will be.

In general, we cannot even assume that the whole cache of chunks for a block resides within a single satellite. Distributing the cache saves memory but also improves parallelism in setting and getting values. A cache miss is not catastrophic as a KV cache can always be recomputed. So, while redundancy is not required for reliability, it can improve latency. The more blocks you can cache, the higher likelihood of cache hits, and therefore the lower the inference time.

The networking model of a 2D-Torus mesh with 4 ISL links from each satellite is called +GRID and illustrated in Figure 2 below (from [7])

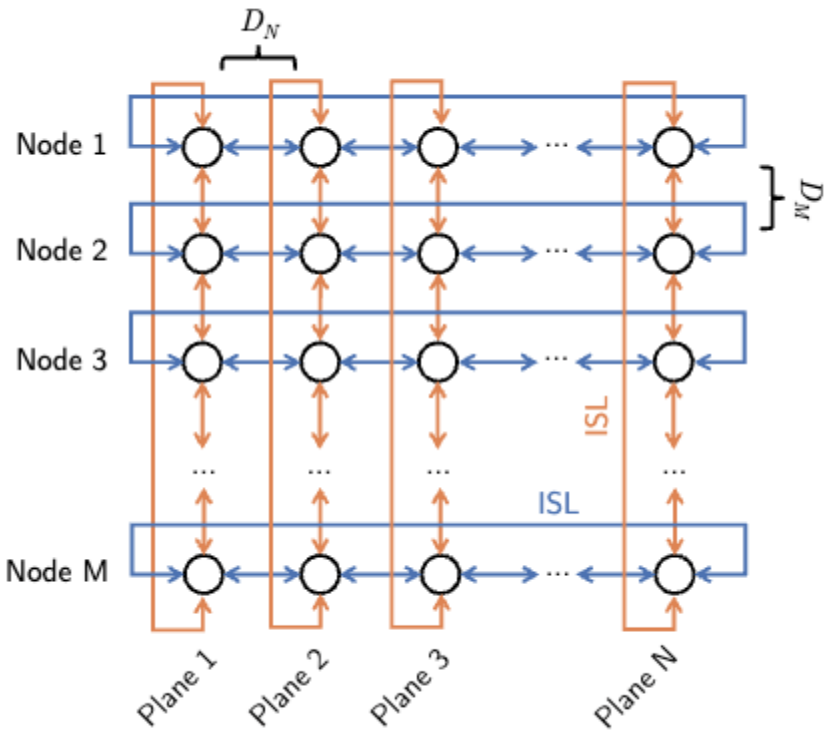


Figure 2. The +GRID networking model, so named because the 4 ISL links from each satellite look like a plus sign. Note that the topology is a torus, so the topmost and bottom-most rows wrap around, as do the leftmost and rightmost columns.

Although the ideal architecture would be to connect and communicate directly with all line-of-sight satellites from the ground directly, the protocol is structured such that all the cache endpoints are within the fewest possible routing hops from the closest satellite, which could be used if the line of sight (LOS) is obstructed.

Interface

The abstract interface provides two operations to obtain a KVC for a prompt (or empty KVC if none was found) and to add blocks in the cache based on a prompt. Both operations operate on a given model and tokenizer. If any parameter changes in the model, the cache is no longer valid. Similarly, a different tokenizer would also invalidate the cache.

```
class KVCManager:
     init(model:LLMModel, tokenizer:LLMTokenizer)
     add_blocks(prompt:String)
     get_cache(prompt:String) ->KVC
```

The KVC returned can be passed into `model.generate` calls to speed up generations, often referred to as past key values. The KVC can be implemented to be memory efficient by trading off accuracy using various quantization techniques.

Chunk to Server Mapping

Servers are virtual satellite destinations, and the number of servers determine how the chunks are spread out in a deployment. The initial chunk is always stored in the current closest LOS satellite. The server ids are then distributed around this point in concentric circles to lay out the chunks.

Migration

Once a new set of satellites are in LOS the chunks need to be migrated for efficient retrieval. This can be done in parallel in each orbital plane.

Rotation Aware Caching

Servers are mapped to satellites left to right, top to bottom within a LOS grid. See Figure 3.

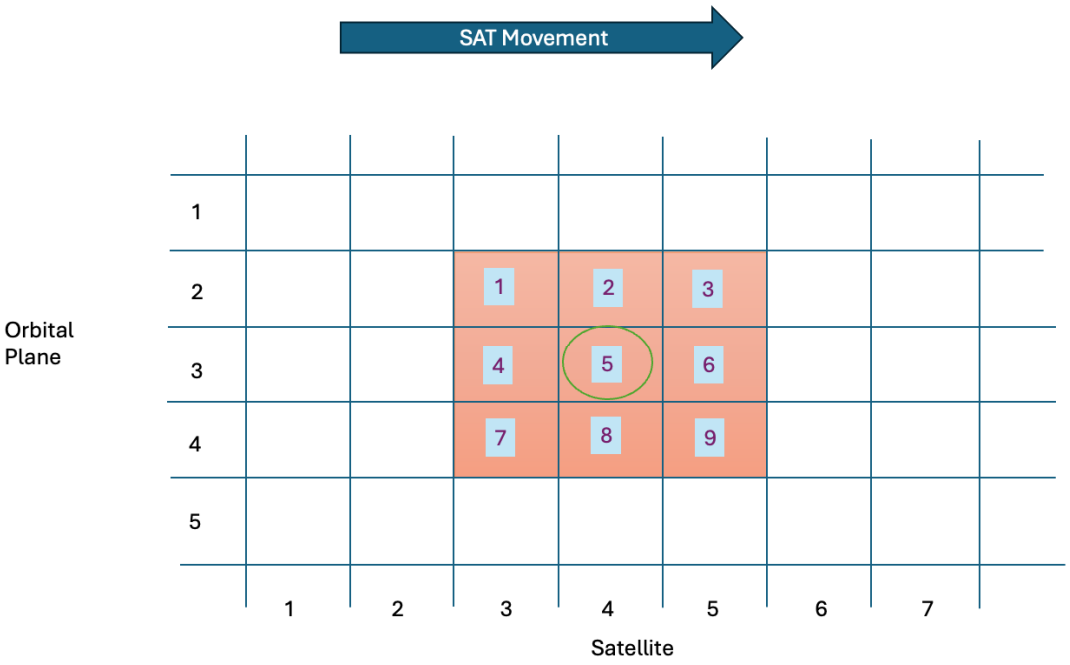


Figure 3. Orange area is LOS. Green circle is the satellite closest to the LLM.

This works best in the use case of a ground hosted LLM that has LOS to a large number of satellites reliably (e.g. 10-20). Migration is done from the column furthest to the right to the column furthest to the left. See Figure 4.

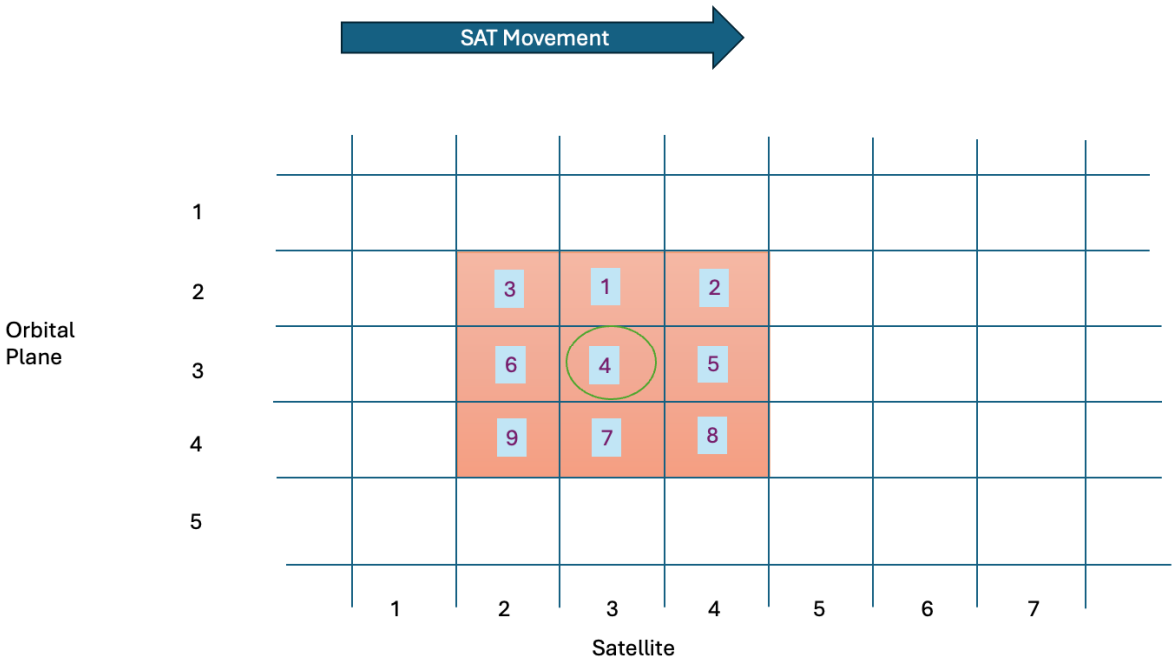


Figure 4. Illustrating migration from the rightmost of the colored columns to the leftmost colored column.

Hop Aware Caching Servers are mapped to satellites in concentric circles starting from a given satellite, as shown in Figure 5. This works best when no migration is necessary and the LLM is hosted on-board a fixed satellite.

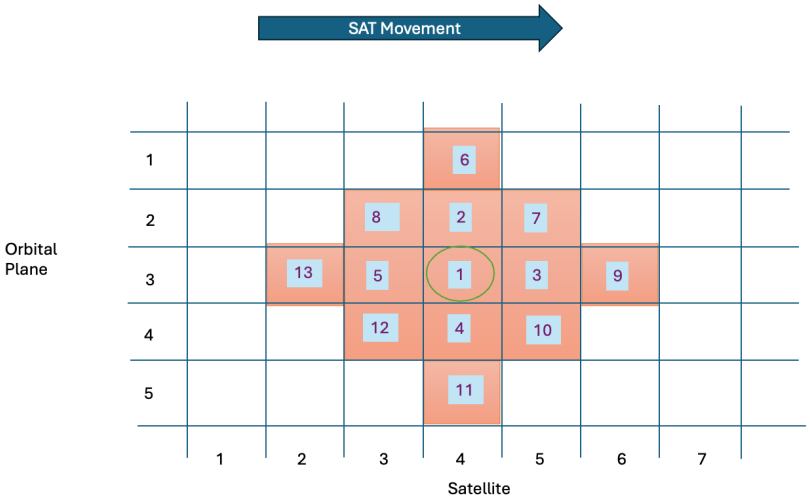
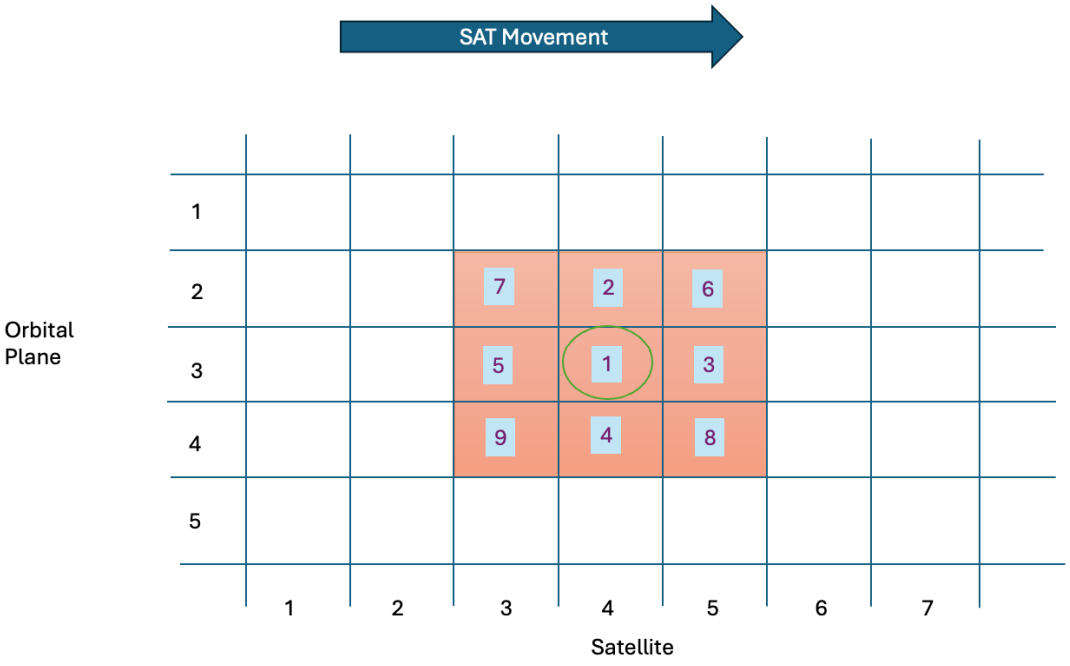


Figure 5. Labeling of concentric circles of servers around the satellite in the center.

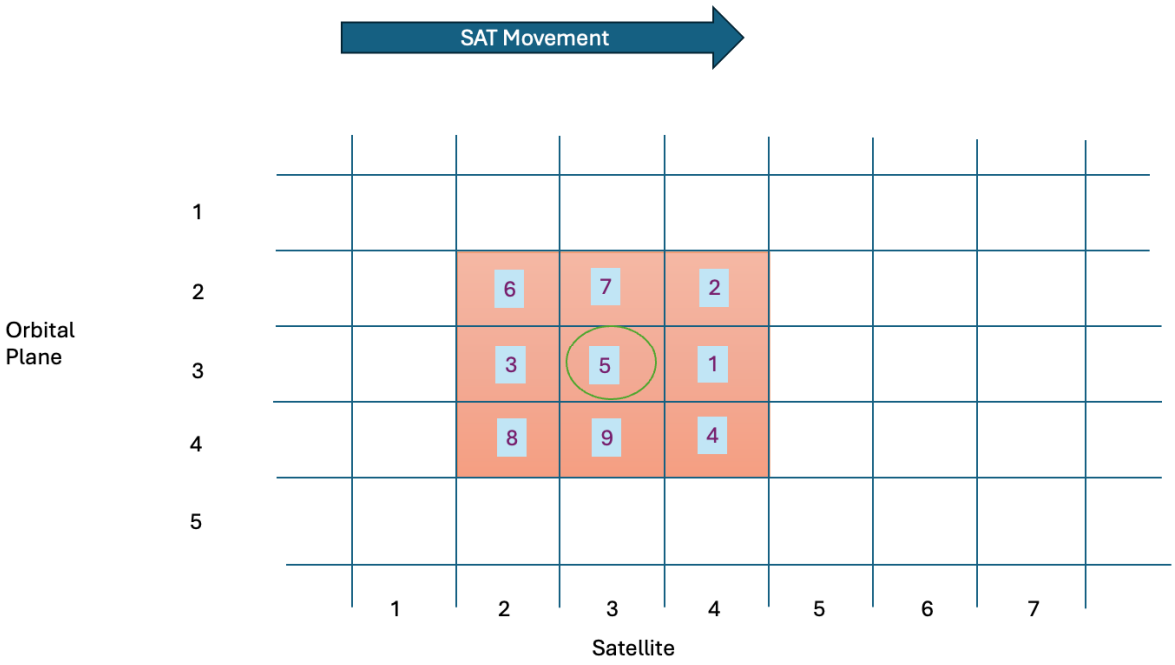
Note that the concentric circles and hops may be logical so that faster horizontal within-plane hops can result in wider horizontal areas.

Rotation and Hop Aware Caching

Servers are mapped to satellites in concentric circles starting from a given satellite and within a LOS grid. This works best in the ground to satellite scenario like the rotation case below but where the satellites cannot be reached reliably wihin a single hop from the ground.



(a)



(b)

Figure 6. Illustrating rotation- and hop-aware caching. (a) Initially, the green satellite (satellite 4 in orbital plane 3) is directly overhead. (b) Given the direction of satellite movement, the column of satellites to its left (i.e., satellite 3 in the orbital plane 3) will be directly overhead in a few minutes. Thus to prevent chunks 6, 3, and 8 from going out of LOS, these chunks are migrated to satellite 2 in orbital planes 2, 3, and 4.

In Figure 6, the orange shaded area represents line of sight (LOS) and the green circle is the satellite that is closest to the stationary LLM on earth. Chunks are stored based on concentric "circles" based on hops with numbering starting in the center, then proceeding clockwise with north first. As the satellites furthest to the east move out of sight their chunks are migrated to the satellites about to enter LOS on the west. In this case Satellite (sat 5,orb 2) migrates chunk 6 to (2,2), 3:(5,3)->(2,3) and 8:(5,4)->(2,4). Note that all these migrations can be done in parallel and there is no harm in the chunk being stored in two satellites for some period of time.

Lastly, we note that if we predict a cache hit on a certain set of chunks at some future time (for example, because of a predictive algorithm), then we can exploit the fact that the set of satellites in the LOS at that future time is known exactly and arrange to make those chunks available on those LOS satellites at that time.

Protocol:

Set KVC:

1. The prompt is tokenized and split into equal-sized token blocks
2. Each block is ordered and hashed based on the previous block hash and the token list in the current block . The initial block has a null hash 0 as the previous block hash.
3. A lookup is done for each block (See Get KVC protocol below) starting at the last block and stopping when a match is found (alternatively we can use a local radix tree, see below)
4. If no match is found for a block the KVC for that block is split into fixed byte chunks
5. Each chunk is mapped into a separate LOS satellite using `chunk_id mod num_los_sats`, the one with the fewest hops store `chunk_id 1`
6. The mapping from server to chunk it is done with the current closest satellite as chunk 1. Then follows a pattern left to right top to bottom in concentric circles. When a satellite is about to exit the LOS region all chunks stored need to be migrated to the satellite about to enter LOS in the same plane.

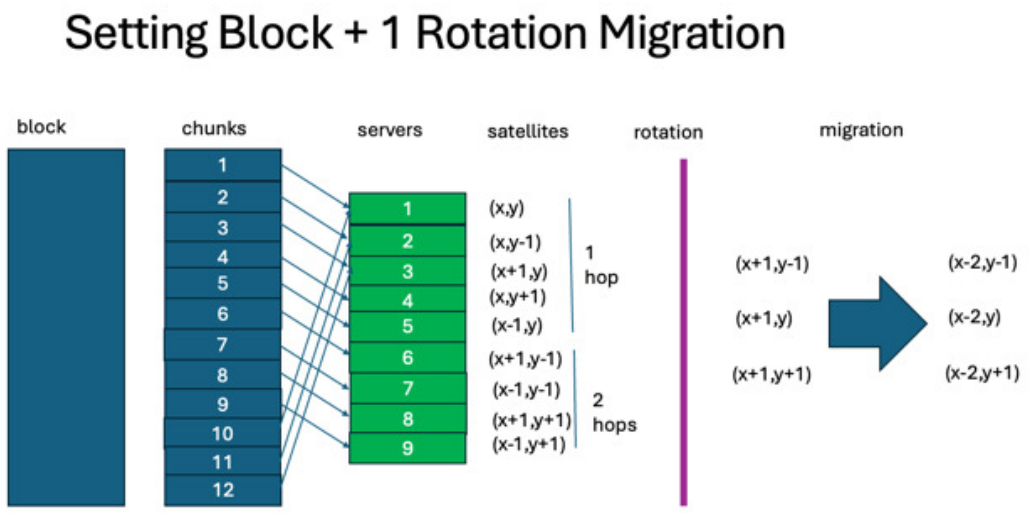


Figure 7. How chunks are created from blocks, mapped to logical server and then satellites, and how the new satellite hosts for a chubnk can be computed after a rotation.

Get KVC:

1. See Step 1 in Set KVC
2. See Step 2 in Set KVC

3. The block list of hashes is searched with a binary search for a hash with chunk 1 within the satellite in LOS with fewest hops (or radix tree)
4. If a match is found the higher ordered half is searched
5. If a match is not found the lower ordered half is searched
6. If the higher ordered half is a single hash and it is not found the search stops with the result that the hash/block was not found and an empty KVC is used in the generation
7. The latest (highest ordered block) match is retrieved and all the chunks for that block are retrieved to reconstruct the KVC to be used in the generation
8. The lookups always start at the nearest satellite. It will return its chunk id and based on that the shift left to right in chunk to server mapping is found and the server for all other chunks can be computer and all chunks can be queried in parallel.

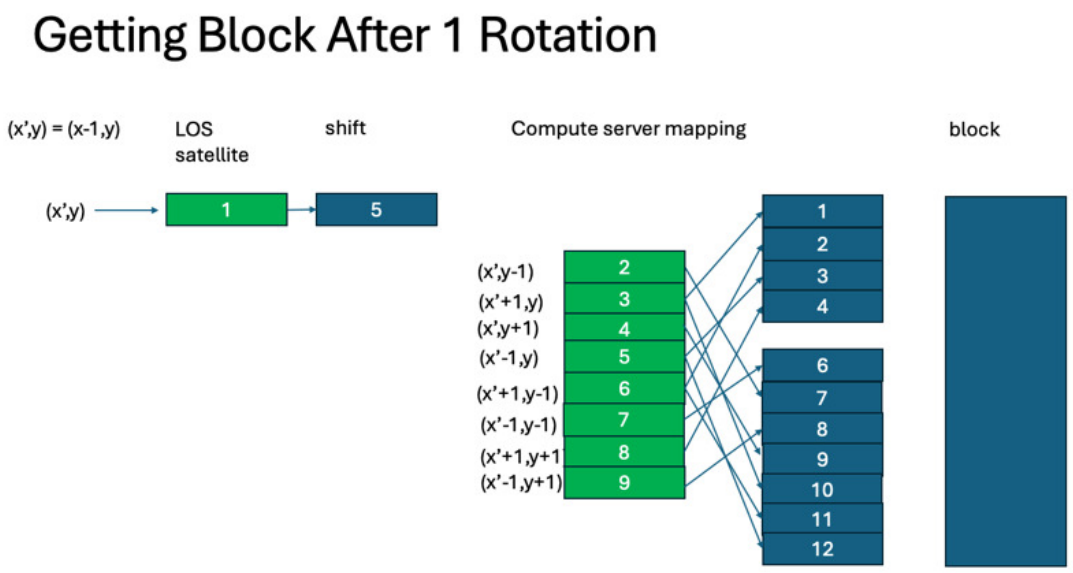


Figure 8. How a block can be recreated for chunks after one rotation. Note that rotations are predicable based on knowing the time of block creation.

Chunk Eviction

When there is memory pressure, the LRU chunk will be evicted to make place for new block chunks. As soon as one chunk is gone, the block it belongs to cannot be retrieved and must be purged. Hence an eviction needs to be propagated to all the satellites holding the same chunks or at least the orbital plane that is used for lookups. If a block only has one chunk, this is not an issue, and it could be a reason to keep the chunk size large as a tradeoff for parallelism in retrieval and storage.

The advantage of the concentric circle of storage of chunks is that all the chunks impacted by eviction are in the direct neighborhood of the chunk initially being evicted, hence a simple gossip broadcast in all directions is sufficient. Alternatively, lazy eviction can be implemented, where the lookup client will issue evictions when chunks in a block are discovered to be missing.

Yet another policy is to do cleanup of chunks that are not complete periodically. When chunks migrate, they can be evicted so there is a natural eviction as part of the rotation synchronization as well.

Local Radix Block Index In the case of transformer KVC caching the ordered blocks resulting from the prompt need to be queried using a longest prefix lookup. The last block in the block list matching will have the cache we are interested in. The lookup may be done sequentially or with some binary search, but as an optimization we propose storing the block keys (not the values) in a Radix Tree (a.k.a radix trie or prefix tree) locally were the LLM resides (on the ground or on-board the satellite).

An efficient lookup can thus be made in this local data structure to find out whether the block exist. The database entry for the block can also store meta data such as total number of chunks and the time of setting the value which would allow the caller to compute where each chunk is currently located without having to query any external satellites.

Eviction can either be done when a local lookup succeeds but the values are not present in the constellation or by propagating eviction broadcasts back to the LLM.

The figure below shows the full end-to-end process:

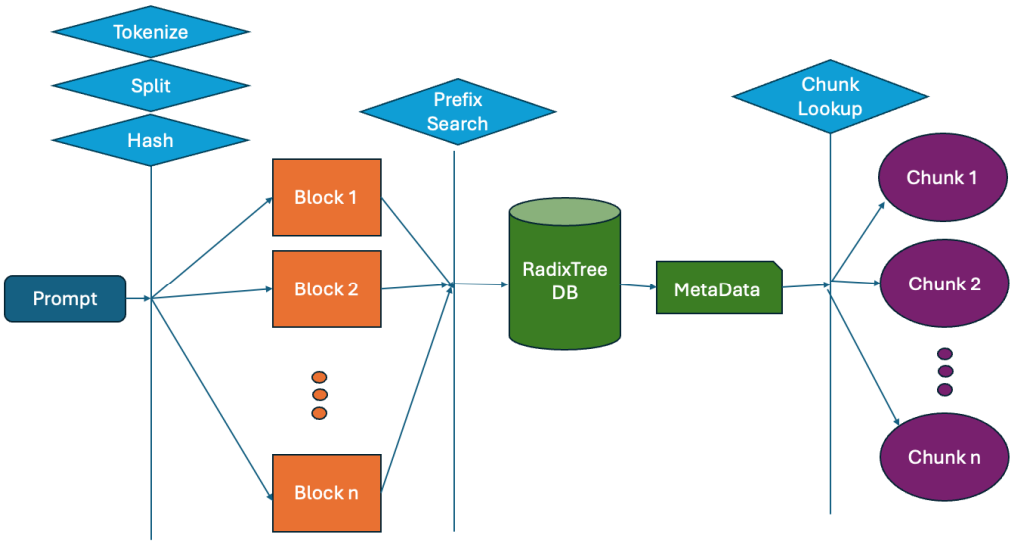


Figure 9. End-To-End process from a prompt to a chunk lookup using a radix tree.

We note that the radix tree lookup is an optional optimization, and it is the only part that is specific to the Transformer (LLM KVC) application. All the other parts of the protocol can be used as a general-purpose in-memory, Key Value Store (KVS) or distributed HashTable for any data where the key can be converted to a string and the value is an arbitrary, potentially large, byte sequence.

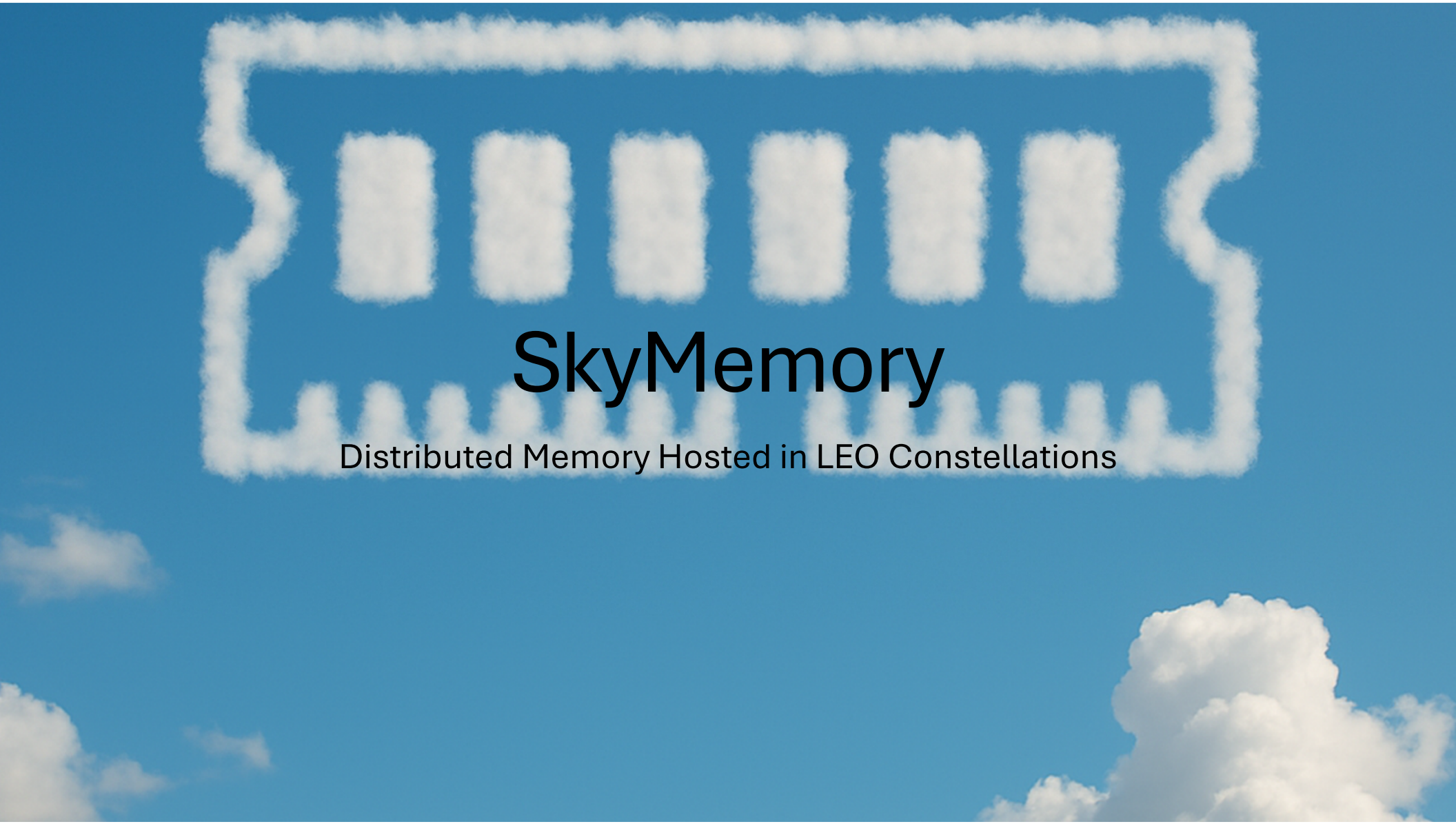
Summary

As AI models get increasingly deployed on the edge, there is a greater need for low-latency access to, and distribution of AI model states on, edge devices. The industry at large is trending toward a memory-first architecture to realize the lowest possible latency for such access. This in turn means that there is a greater need than ever for shared memory platforms. As we send an ever-increasing number of high-capacity satellites into orbit with free space optics links to support communication and earth observation use cases, LEO satellite constellations become an untapped resource for distributed memory. The present invention proposes SkyMemory a solution for Transformer Key Value caching to optimize LLM inference to deliver on that vision. As network operators (both our members and other telcos) begin to deploy so-called “AI factories” or AI distribution points in their plant to host or provide AI services at the enterprise edge, this invention will allow for faster inference and serving more queries and customers in a more efficient manner.

NVIDIA Jetson Nano GPU Prototype

We hosted a 1B parameter LLM (TinyLlama/TinyLlama-1.1B-Chat-v1.0) on an NVIDIA Jetson Nano 8GB GPU and fed it a ~250-character prompt as context. It resulted in 4x 128 token blocks of about 2.9MB each (with optimum-quanto 8bit quantization). The blocks were split into 6k chunks before storage and retrieval. We used 10 LOS cFS [8] satellites to stripe the chunks across. This resulted in a 30 token generation speedup from 6.2s to 4.9s or 21% when running the generation without and with the cache respectively. For a HQQ quantizer the speedup was about 24%.

Quantization	No KVC (seconds)	KVC (seconds)
Optimum Quanto	6.2	4.9
HQQ	10.2	7.8


SkyMemory
Distributed Memory Hosted in LEO Constellations

Why?

MEMORY EXPANSION INTO EXTERNAL SATELLITES

INCREASED CACHE HITS

IMPROVED SPEED OF INFERENCE

Value to members

- Members are upgrading their gateways to support AI optimizations and to host 3rd party AI applications (AI Factories)
- Caching is critical for the performance of inference at scale (NVIDIA Dynamo claims 30x speedup for some workloads)
- Members are already forming partnerships with Satellite operators (Rogers,GCI)
- AI Factory value-proposition compared to Cloud AI is low latency. Caching is critical to delivering low latency at scale

Objectives

- Minimize latency (hops) to set items
- Minimize latency to get items
- Account for orbital rotation
- Account for LOS
- Allow item localization given orbital shift (deterministic placement)
- Use small packets for less memory footprint and increased parallelization
- Minimize message communication volume
- Efficient continuous state migration (to maintain lop-hop behavior despite rotation)

Distributed LLM Inference Optimization

- Pre-fill caching key to trade off computation with memory storage
- Pre-computed vectors stored in Key Value Cache (KVC)
- KVC may be stored in a memory layer of GPU, CPU, RDMA, SSD etc memory
- Proposal: Add LEO edge as a memory layer

Why LEO?

- Low latency access from anywhere on earth
- Large network of high-capacity refrigerator/car sized compute/memory devices (capable of supporting Mobile Cell Tower in Space)
- Free Space Optics Inter Satellite Links for efficient speed-of-light synchronization
- High Radix Routing

Feasibility

	Latency
CPU	10-15 nanoseconds
GPU	50-100 nanoseconds
RDMA	2-5 microseconds
SSD	20-200 microseconds
HDD	2-20 milliseconds
NAS	30-40 milliseconds
LEO (current)	20-50 milliseconds
LEO (theoretical with laser)	2-4 milliseconds

ISL Starlink Constellation Latency Example

Starlink h: 550,000m - M (sat/orb): 22 - N (orbs): 72 - r_E: 6371,000m

Constellation: 1,584 satellites

- Intra-Orb

$$D_M = (r_E + h)\sqrt{2(1 - \cos(\frac{2\pi}{M}))}$$

D_M=1,969,922m

L_{sat} = 1,969,922/299,792,458 = 6.6 milliseconds

- Inter-Orb

D_N= 603,780m

L_{orb} = 603,780/299,792,458 = 2 milliseconds

What is a KVC?

- Cache to avoid re-computing LLM output vectors for previously seen input vectors
- Valuable for large frequently used contexts in prompts
- Prompt -> Token Sequence -> Ordered Token Blocks
- Key Feature: Block is mapped to KVC
- KVC -> Serialized Byte Chunks

- Chunking allows parallelism and efficient storage/retrieval

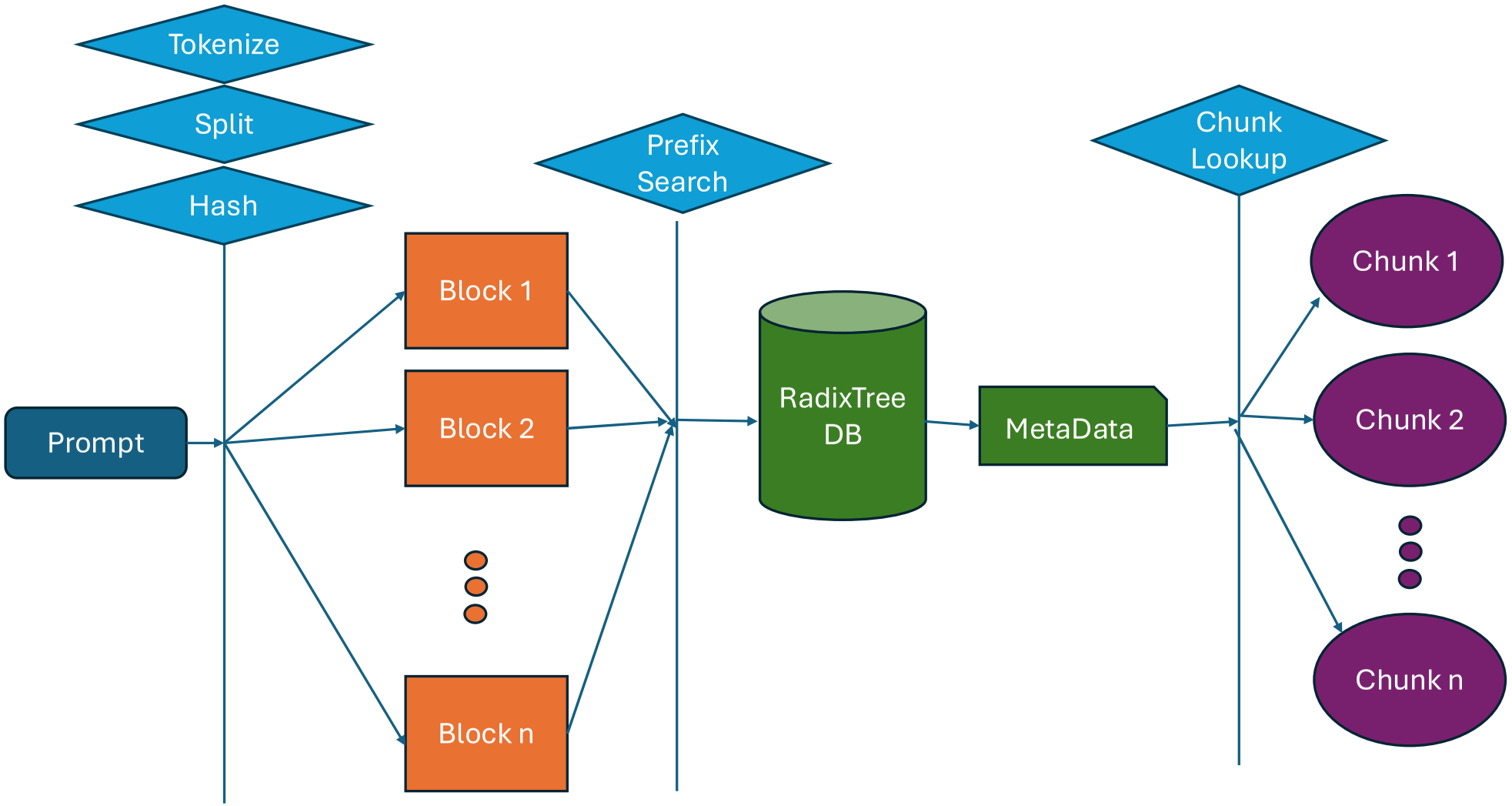
How can we efficiently host a KVC in a LEO Constellation?

- For ground LLMs
- For on-board LLMs

- Efficient Parallelism
- Efficient Storage
- Both Set and Get Heavy Workloads
- No replication/reliability needed
- Needs to be Hop and/or Rotation Aware

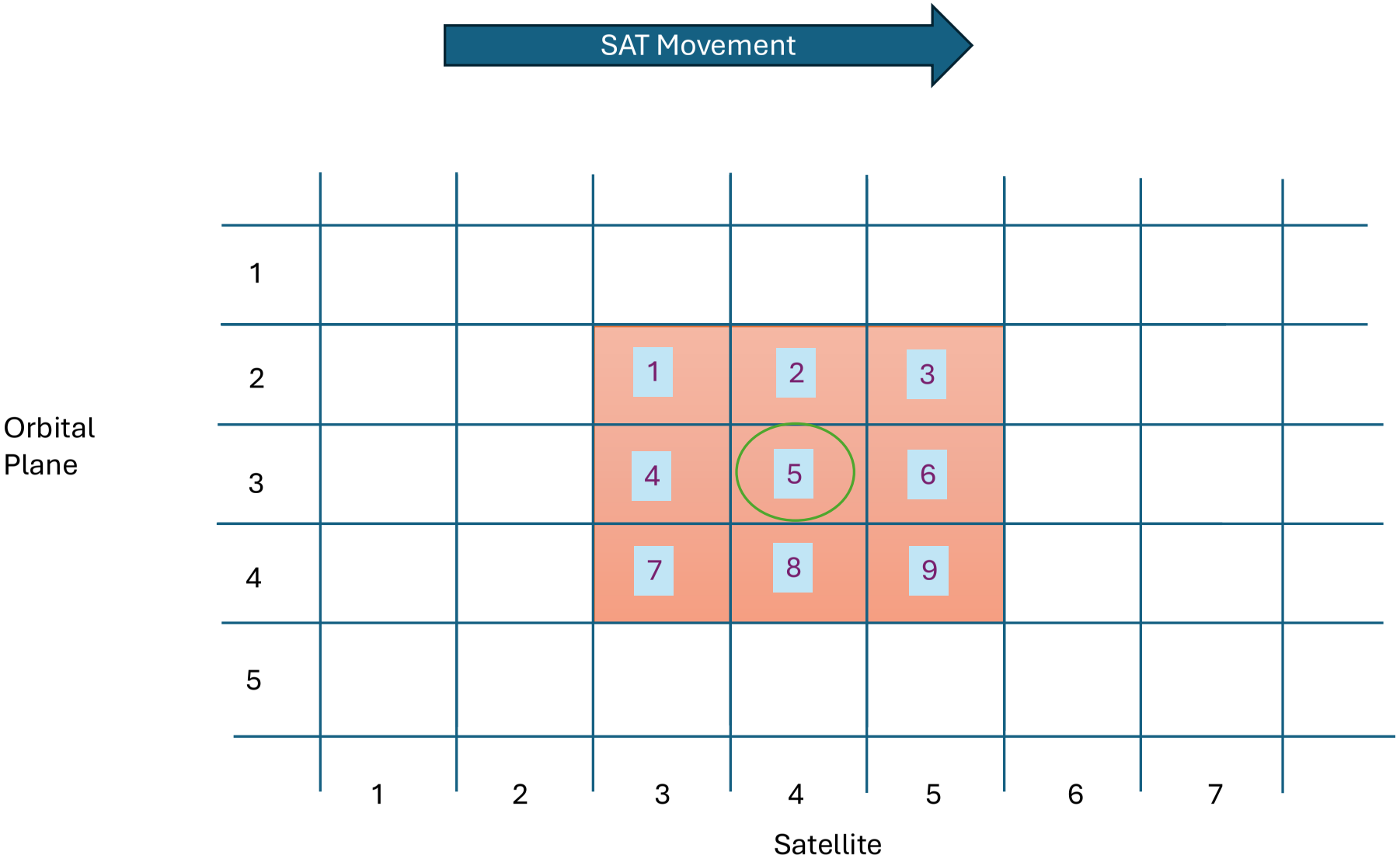
Block Lookup

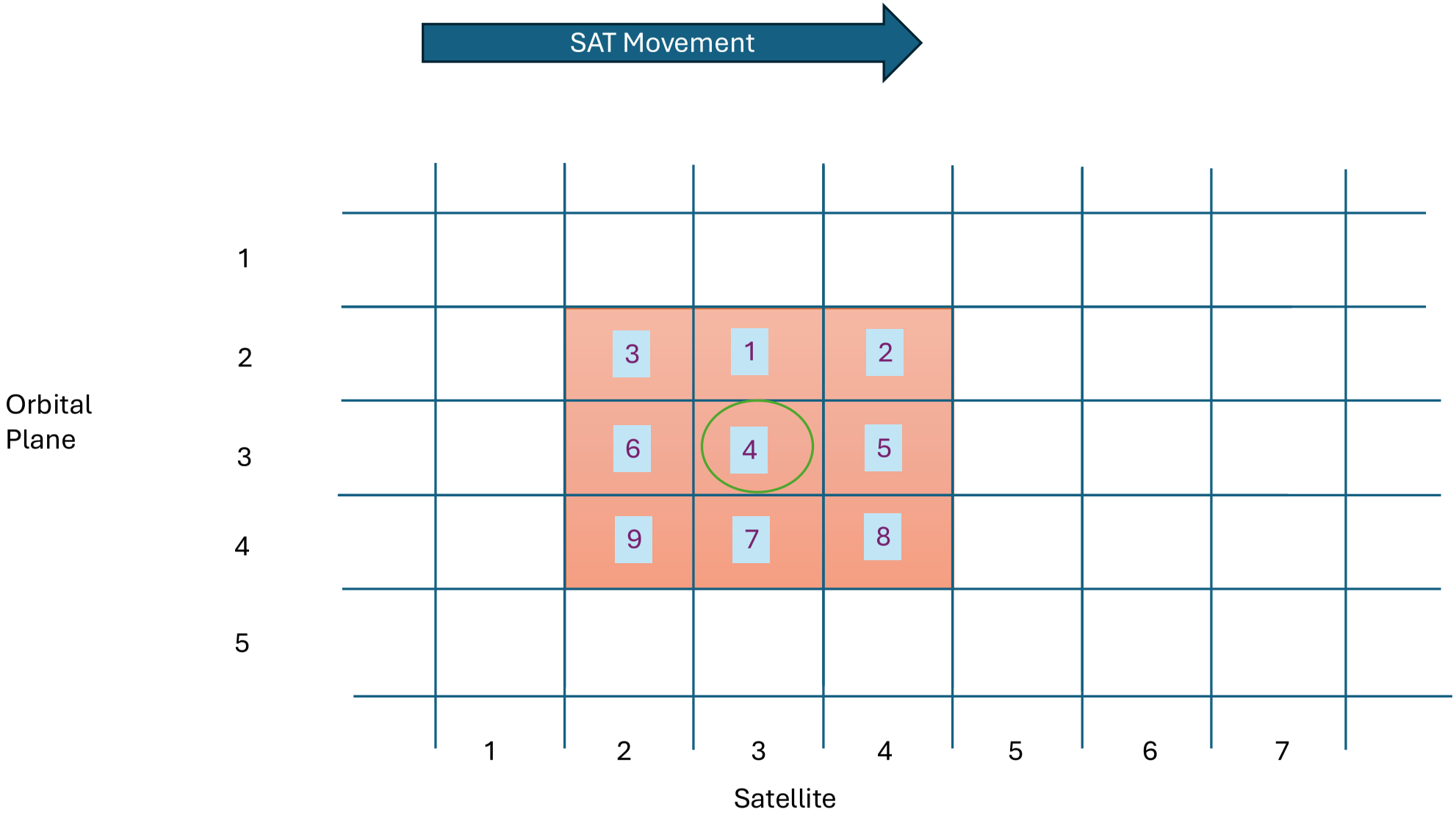
- An Ordered Block List with Hashes representing tokens is input to the lookup
- The hashes can be converted into a key in a radix tree
- A query into the radix tree finds the longest prefix match
- The radix tree can store meta data of the hashes like the number of chunks and the shift (time of first storage)
- If a matching block is found the locations of the chunks will be computed, based on the total number of chunks and time shift, from the external LEO satellites described next


Tokenize
Split
Hash
Prefix Search
Chunk Lookup
Prompt
Block 1
Block 2
Block n
RadixTree DB
MetaData
Chunk 1
Chunk 2
Chunk n

Rotation Aware

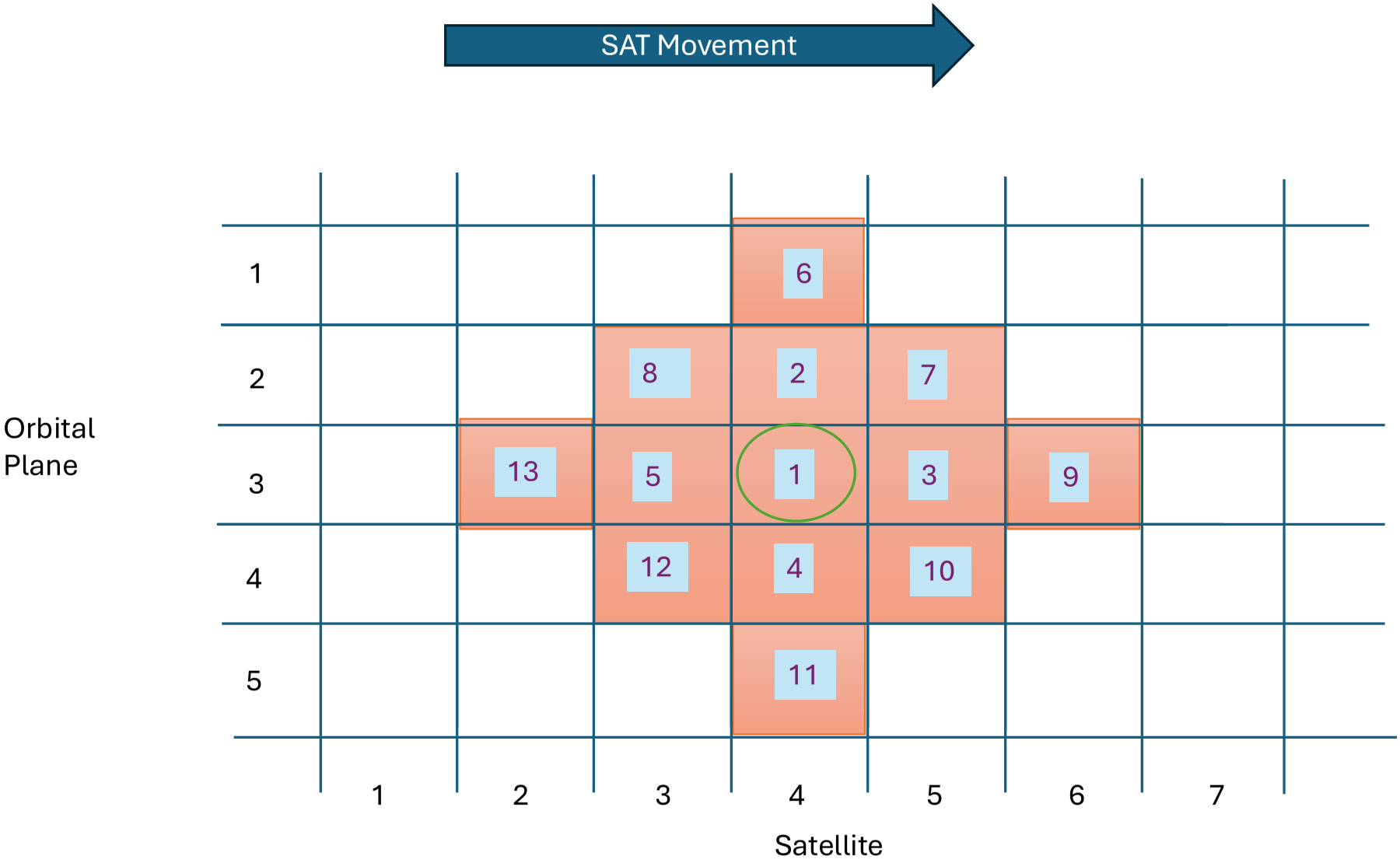
For 2D-Torus communication from Ground -> Satellite when only a single satellite is in LOS


SAT Movement
Orbital Plane
1
2
3
4
5
1
2
3
4
5
6
7
Satellite


SAT Movement
Orbital Plane
1
2
3
4
5
3
1
2
6
4
5
9
7
8
1
2
3
4
5
6
7
Satellite

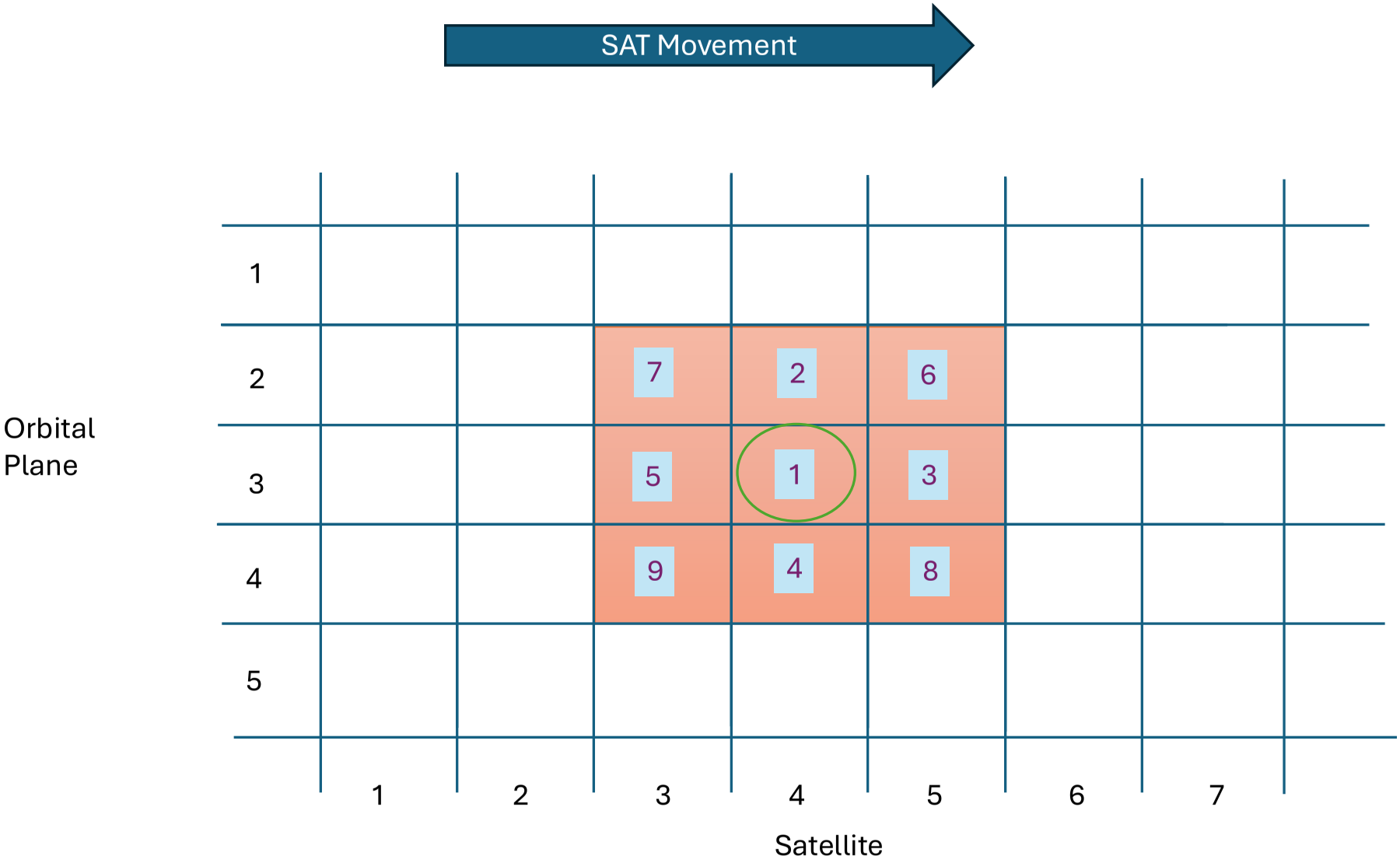
Hop Aware

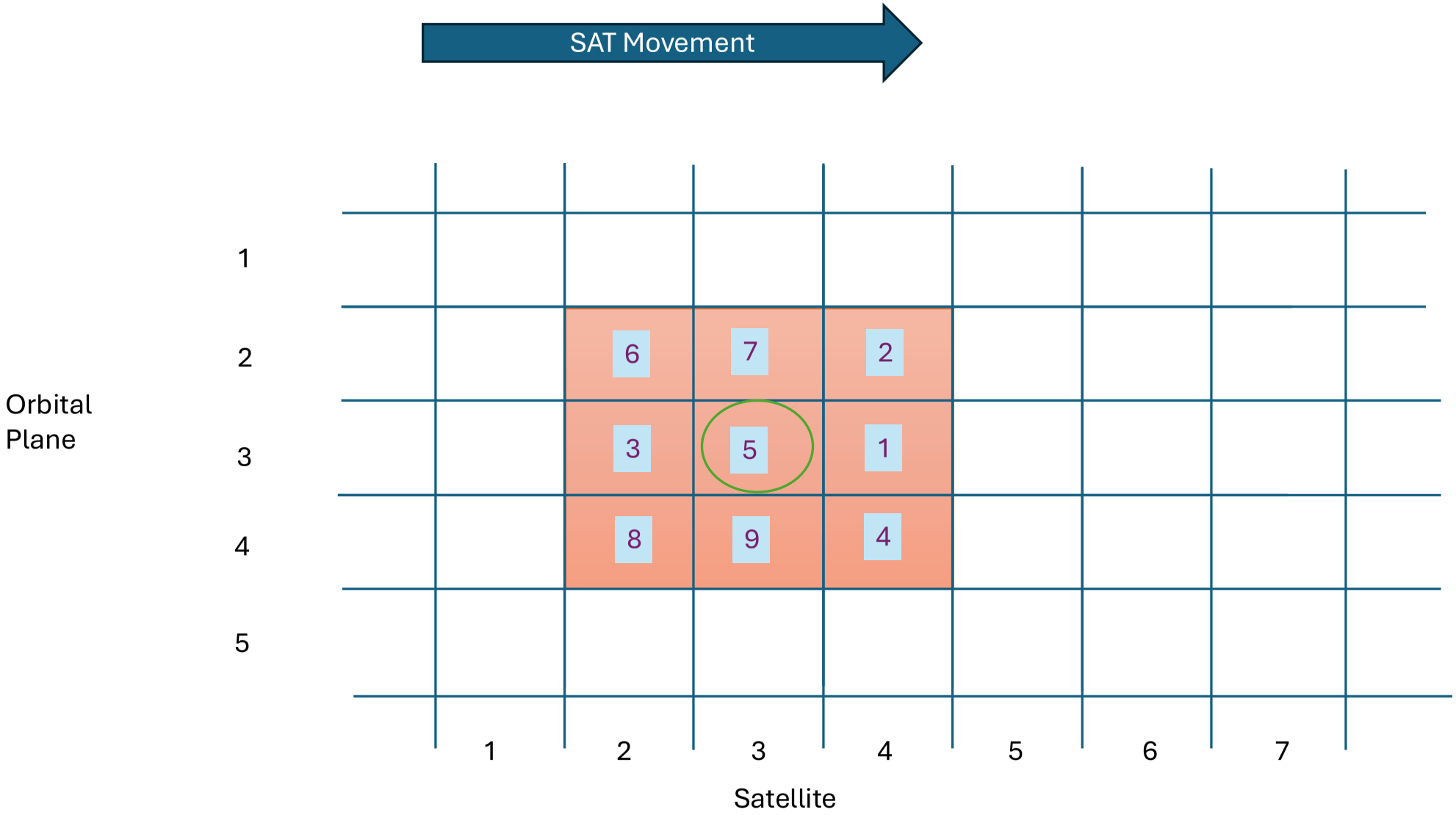
For 2D-Torus communication from Satellite -> Satellite


SAT Movement
Orbital Plane
1
2
3
4
5
6
7
8
2
7
13
5
1
3
9
12
4
10
11
1
2
3
4
5
6
7
Satellite

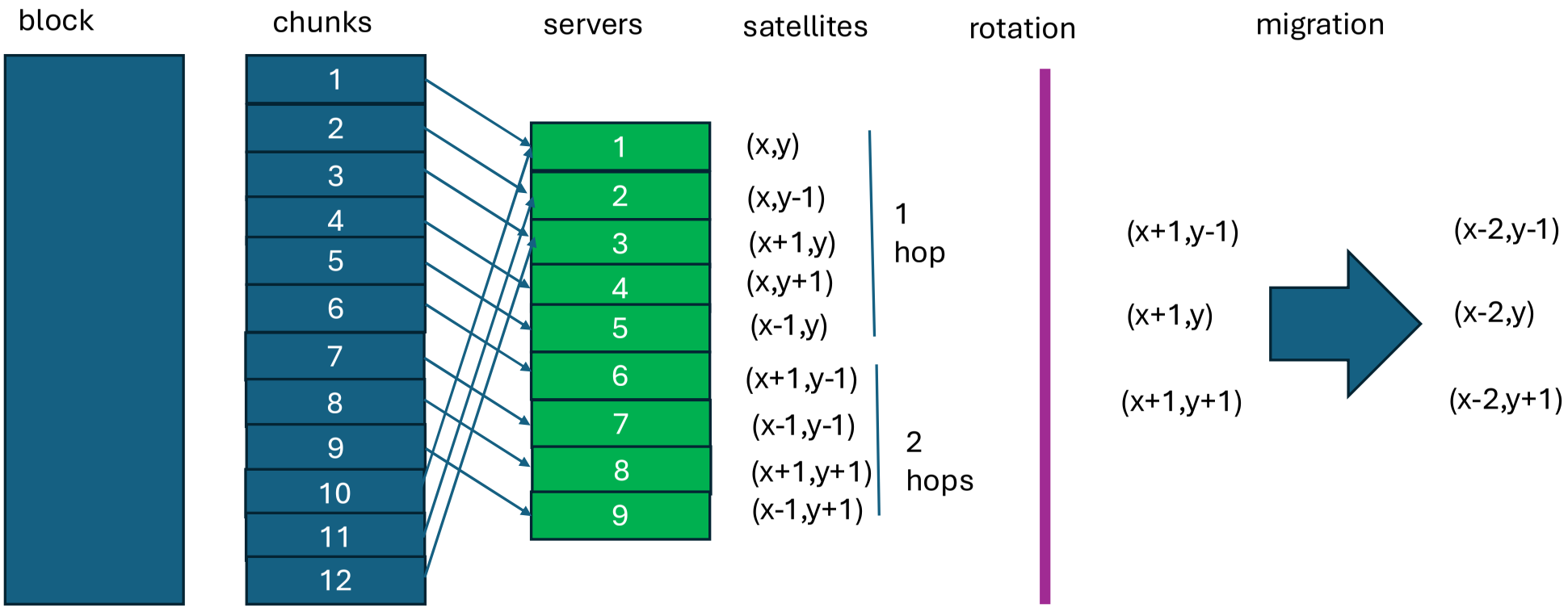
Rotation and Hop Aware

For Direct LOS communication from Ground -> Satellite

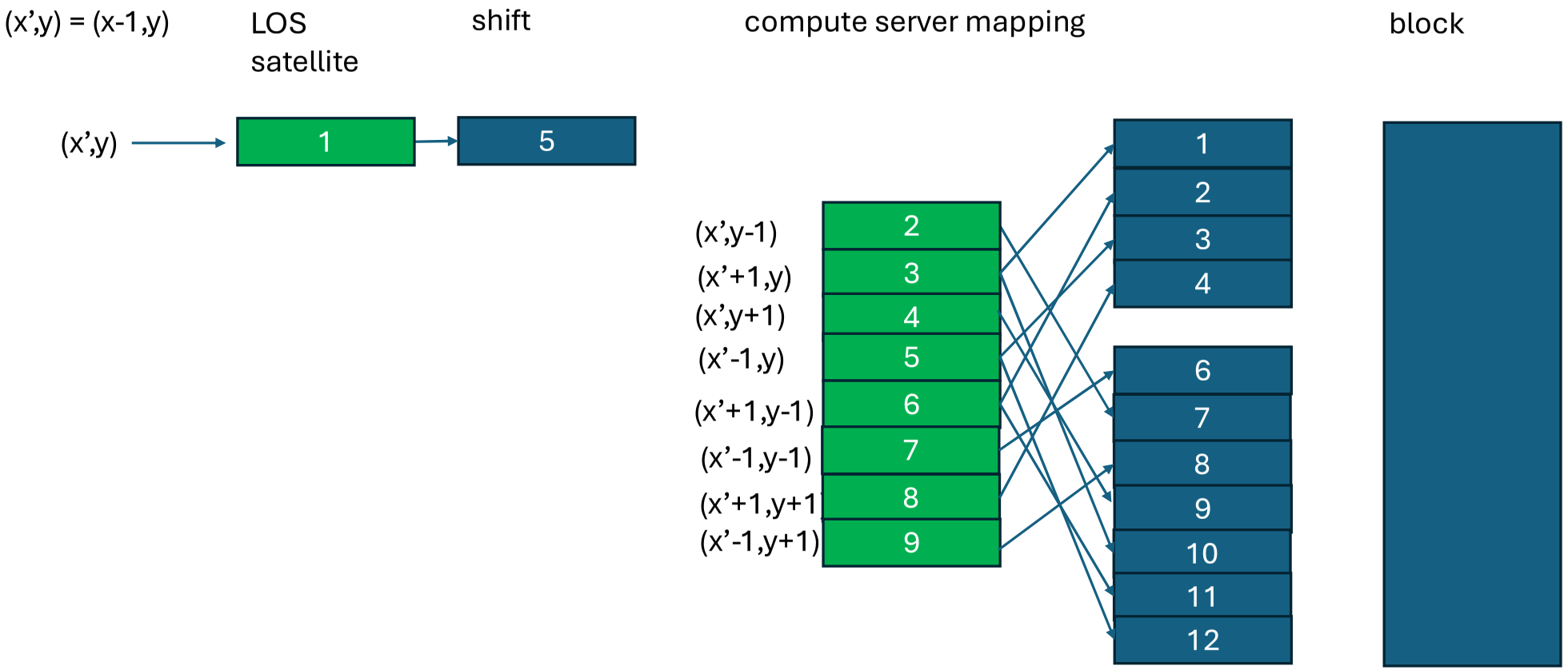

SAT Movement
Orbital Plane
1
2
3
4
5
7
2
6
5
1
3
9
4
8
1
2
3
4
5
6
7
Satellite


SAT Movement
Orbital Plane
1
2
3
4
5
6
7
2
3
5
1
8
9
4
1
2
3
4
5
6
7
Satellite

Setting Block + 1 Rotation Migration



Getting Block After 1 Rotation



Example Use with Huggingface Transformers

```
kvc = KVCBlock()
inputs = tokenizer(prompt , return_tensors="pt”)
out = model.generate(**inputs, past_key_values=kvc.get_cache(prompt))
kvc.add_blocks(prompt)
```