1. **Spherical Octree Partitioning**

P5, P6, P7 & P8

A spherical octree is a hierarchical spatial index that recursively divides the sky sphere into 8 regions at each level, based on data density.

Why Spherical Octree
- **spatial locality**
- **hierarchical refinement**
- **uneven data density handling**
- **distributed load balancing**

Level - 0 Partition

0 → [0°,180°], ϕ → [0°,90°]
0 → [180°,360°], ϕ → [0°,90°]
0 → [0°,180°], ϕ → [-90°,0°]
0 → [180°,360°], ϕ → [-90°,0°]
= split each hemisphere into 4 quadrants

So we get 8 primary Spatial partitions:

P0, P1, P2, P3, P4, P5, P6, P7

Why Spatial Partitioning First?

Space observations are inherently spatial and unevenly distributed.
A partitioning strategy must preserve spatial locality, support hierarchical refinement, and allow scaling of dense and sparse region s.

Level-0: Coarse tiling of the celestial sphere
Deeper levels: Higher resolution only where observations exist (assumption for futuer need)

Design Goals
- Preserve spatial locality for range queries
- Adaptive refinement in dense regions (galaxies, clusters)
- Independent scaling of sky areas
- Minimize cross-node communication
- Enable query pruning at partition level

Note
Partitions are **logical sky regions (WHAT data)**, not physical machines.

Technology Mapping
- **Ingestion Router: Kafka + Flink**
- **Metadata Store: Redis / DynamoDB**

Pseudo Code
```
# Flink spatial routing
stream
  .map(event -> addLatLon(event))
  .map(event -> event.partition = octree(lat, lon))
  .sink(kafka_topic_by_partition)
```

2.Recursive Subdivision

Each partition can be subdivided into 8 sub-partitions based on data volume and query patterns. Depth represents resolution, not fixed struct (Cn go more based on resolution):

lets assume P3 partition
P3.0
P3.1
P3.2
.
.

Depth = resolution
This is hierarchical partitioning.

How many partition do we need ?
If partition Size > threshold -> split it
If region is sparse -> keep shallow

How many shrds do we need?
Because one machine cannot handle all data.
So we distribute data across machines **(horizontal scaling and parallelism)**

Assumptions of how to calculate shards
Total data = 1 PB
- ***One node can handle ~ 10 TB***
  Number of nodes = 1 PB / 10 TB = 100 nodes
  Shards = 10 × number of nodes

Nodes = 100
Shards =1000

Partition = WHAT data? (logical region)
Shard = WHERE data? (physical distribution)

Model to assume where the data sits

Logical Table: Observations
↓
Spatial Partitioning (Octree) (Meaningful data region)
↓
Partitions (P0, P1, P2...)
↓
Shards inside each partition (Physical split)
↓
Nodes (machines)

Technology Mapping
1. Sharding: Kafka partitions / Cassandra tokens
2. Cluster: Kubernetes nodes
3. Monitoring: Prometheus

3.Partition Key Design

Partition Function Principles
- **Deterministic mapping from (lat, lon) -> partition**
- **Depth chosen adaptively based on density**
- **Must preserve spatial locality**
- **over time for rebalancing**

Design Objectives
- **Nearby observations map to same partition**
- **Balanced load across regions**
- **Minimal cross-partition queries**
- **Fast routing during ingestion**

partition_id = f(latitude, longitude, depth)

Pseudo Code
```
def getPartition(lat, lon, depth):
  p = baseR(lat, lon)
  for i in range(depth):
    p = subdivide(p, lat, lon)
  return p
```

Technology Mapping
1. Geo Library: Google S2 or Uber H3
2. Config: Etcd

4. Logical Data Model Characteristics
- **High dimensional**
- **Extremely sparse**
- **Append-heavy**
- **Analytical workloads dominate**

Observation(
observation_id,
timestamp,
telescope_id,
lat,
lon,
depth,
features[10000]
)
.
.
.

Technology Mapping
- **Schema Registry: Confluent Schema Registry**
- **Format: Avro / Protobuf**

5.Physical Data Model (Initial assumption)

Sparse Representation

Only non-zero values are stored using triplets: (observation_id, feature_id, value)

Benefits
- **Storage proportional to non-zero values**
- **Eliminates 99%+ zero overhead**
- **Natural fit for distributed processing**
- **Efficient compression**

(observation_id, feature_id, value)
102938, 12, 0.87
102938, 209, 3.2
102938, 900, 10.01

Technology Mapping
- **Sparse Store: Cassandra / HBase**
- **Column Store: Parquet / ORC**

Pseudo Code
triplet = (obs_id, feature_id, value)
spark.write.parquet(triplet)

6.End-to-End Architecture Diagram

Telescopes

Telescopes generate raw signals.

raw_signal = telescope.capture()
features = extract_sparse_features(raw_signal)

Why edge processing? (Not Mandatory can be eliminated to preserve the Actual source of truth)
Raw telescope signals are filtered near the source to extract sparse features and reduce data volume before ingestion.

Edge Signal Filtering

Distributed ingestion workers:

worker_loop():
  while true:
    batch = queue.pull_batch()
    partition = getPartition(batch.lat, batch.lon)
    route_to_partition(partition, batch)

Global Ingestion Layer

Technology Mapping
- **Ingestion: Kafka**
- **Stream Processing: Flink**
- **Hot Store: RocksDB / Cassandra**
- **Cold Store: Parquet on S3/HDFS**
- **Compute: Spark / Trino**
- **Index: Bloom (Cassandra)**
- **Search: ClickHouse / Druid**

Spatial Partition Router

Pseudo Code

```
# Kafka ingest
kafka -> flink -> hot_store

# Background ETL
spark.read(cassandra)
    .toColumnar()
    .write.parquet(s3)
```

This is hierarchical sharding where we route the event to specific paritiion to the actual storage node

```
function route_to_partition(partition_id, event):
    shard = hash(event.observation_id) % N
    storage_node = shard_map[partition_id][shard]
    storage_node.write(event)
```

How do reads work

Step 1: Find partitions.
partitions = findPartitions(region_R)

Step 2: Find shards in those partitions.
shards = getShards(partitions)

Step 3: Read in parallel.
parallel for shard in shards:
    read(shard)

Distributed Sparse Storage

Hot store & Cold Store

Hot Store (recent sky data timestamp basad filter)
- **row-based sparse store**
- **low latency**
- **in-memory + SSD**

Cold Store (universe archive)
- **columnar sparse store**
- **extreme compression**
- **object storage**
- **Historical archive**
- **High compression**
- **Object/HDD storage**

Pseudo write path:

```
write(event):
  hot_store.append(event)
  async cold_store.append(event)
```

Raw Data example (Initial Data model)

```
row_id | col1 | col2 | col3 | ... | col10000
1      | 0    | 0    | 5.2  | ... | 0
2      | 0    | 0    | 0    | ... | 0
3      | 1.1  | 0    | 0    | ... | 0
```

Sparse Representation
Row 1: [0,0,5.2,0,0] -> {row_id=1, col_id=3, value=5.2}

After the need for analytics, the columnar format is needed to perform aggregation.

Why Column-Oriented Sparse Storage
1. Enables fast aggregations
2. Reads only required columns
3. Skips zeros entirely

Column-Oriented Sparse Storage

Column 3:
row_ids = [1, 100, 2050, ...]
values = [5.2, 7.1, 0.9, ...]

Column 209 (Brightness):
row_ids = [100, 5000, 900000]
values = [3.2, 1.1, 7.5]

Inside each partition + shard, we store data in segments.
Segment Structure -> immutable block containing column block, metadata, row index & bloom filter

Why immutable -> simplified concurrency, compression, replication

Segment -> ColumnBlock(col_id=1)
``````Metadata
``````BloomFilter

```
Segment {
  row_index: compressed list of observation_ids
  feature_blocks: map<feature_id, compressed_values>
  metadata: SegmentMetadata
  bloom_filters: BloomFilter[]
}

FeatureBlock(feature_id):
  row_ids = [101, 203, 999, ...]
  values  = [0.87, 3.2, 1.1, ...]
```

Above is compressed sparse column storage. It is optimal for sparse, high-dimensional data because it stores only non-zero values and enables fast column scans.

**Metadata + Index + Bloom Filters**

**Why Metadata?**
1. Spatial bounds
2. Time ranges
3. Columns present

**Each segment has metadata:**
1.partition_id
2.time_range (min, max time)
3.spatial_range (min lat, min long, max lat, max long)
4.columns_present

**Distributed Compute Engine**

**In the columnar storage point lookup are possible by indexing the row_id**

**Step 1 - Find partition**
partition = getPartition(lat, lon)

**Step 2 - Find shard**
shard = hash(row_id) % num_shards

**Step 3 - Find segment using index**
row_id -> segment_id

**Step 4 - Read sparse columns**
for column_block in segment:
    if column_block.contains(row_id):
        row[col_id] = column_block.get_val(row_id)

**step 5 - Reconstruct the row from column**

**Indexing Strategy**
**1. Partition index (Spatial) -> (lat, lon) -> partition_id (partition lookup)**
**2.Shard Index (Hash) -> row_id -> shard_id (Shard lookup)**
**3.Sort Index (segment-level) -> event timestamp (time lookup)**

**Bloom Filters**
Because scanning petabytes of data is impossible. Bloom filters allow probabilistic skipping with minimal compute .

**Pseudo Code**

```
if !bloom.contains(feature):
  skip(segment)

df = spark.read.parquet(segment)
df.groupBy(col).agg(avg())
```

**Query / Analytics**

**Query Flow**
Find all observations in sky region P3 with feature_209 > 2.0 in the last 24 hours.

**Step 1 - Partition pruning**
partitions = spatial_index.lookup(lat, lon)

**Step 2 - Time pruning**
segments = time_index.filter(last_24h)

**Step 3 - Bloom filter skipping**
segments = filter_by_bloom(segments, feature_209)

**Step 4 - Parallel execution**
parallel for shard in segments:
  result += scan_sparse_data(shard)

**Step 5 - Aggregation**
merge(results)

**Technology Mapping**
- **Spark SQL**

**Pseudo Code**

```
# Aggregation query
spark.sql("""
SELECT avg(brightness)
FROM observations
WHERE partition='P3'
""")
```

**7. Complexity Analysis (Mathematical Analysis)**

Storage Complexity
- Dense model: O(N × M) -> pretty huge.
- Sparse model: O(K) where K <<< N×M

Query Complexity
O(log P + S)
P = partitions
S = segments scanned after pruning

Practical Impact
- 100-1000× storage reduction
- Orders of magnitude fewer I/O operations
- Analytics scales with data relevance, not total size