Exabytes Partition Map Design

Exabyte Scavenger will manage hundreds of petabytes of data in a key-value store. The store is organized around partitions. The design limits are for about 100M partitions on 1M machines, which is about 20x larger than each Data Center opportunity today if partitions remain at 10 GB. Every client for this storage service will have a local “broker” which keeps track of the partition map and routes requests to the correct machine.

The Brokers must actively manage their partition maps. They work with a quorum of supervisors to make decisions about availability or capacity problems which may cause partition map changes, and they work with a gossip network to distribute updates to all Brokers with sub-second delays.

Incidentally, if the 1 Exabyte size is reached we anticipate growth will come from:

1. Larger partition sizes
2. Subsetting the sets of machines to keep Brokers smaller
3. Sharing one broker per server with multiple tenants to allow more than 100M partitions to be an acceptable overhead
4. Other clever ideas

We are confident the actual growth mechanism will best be chosen not now, but later after actual growth and use patterns are known.

# Architecture

The **EB Library**, or service broker, runs on clients of Exabytes Store, hiding the RPC detail from client code. This library keeps track of partition map, and knows which EB server to contact to fetch data. It reports errors and long latencies of EB server to EB manager, as a fault detection mechanism

EB Manager

EB Manager

EBS Server

µ-part

µ-part

µ-part

µ-part

µ-part

µ-part

…

Client

EB Library

EB Server

µ-part

…

µ-part

µ-part

µ-part

µ-part

µ-part

EBS Server

µ-part

µ-part

µ-part

µ-part

µ-part

µ-part

…

EBS Server

µ-part

µ-part

µ-part

µ-part

µ-part

µ-part

…

EB Manager

Control

Data

Each micro partition is replicated to 3 servers: one primary, two secondary. At any time, a server is the primary server for some partitions (we call them *primary partitions* for this server) and secondary server for some other partitions (*secondary partitions* for this server). Primary partitions are stored on SSD, secondary partitions on HDD.

Assuming each server has 1TB of SSD space, which store 100 primary micro-partitions, 10GB each. So we need 2TB of HDD space to host 200 secondary micro-partitions. Each server should handle 3 TB of data.

Under normal conditions, EB library contact the primary for all read and write requests. In the case of primary timeout, the EB library has the option to read from secondary, if the application can tolerate a slight data staleness. Writes, however, have to be handled by primary. Replication group configuration must be propagated to related EB servers, all EB managers and libraries.

# Partition Mapping and Server Chains

In order for the broker to find the right machine for a key, it should have the hash function that maps the key to the partition, together with a partition map. The partitions in turn map onto server chains, where each chain is a collection of K machines. Each partition is assigned to just one of the servers in that chain. When that machine fails, all partitions assigned to that machine will be assigned to some other machine in the chain.

## Server Chain Rules

The server chain has K machines, and every machine belongs to just one chain. Chains need to be balanced in capacity, and diverse in location to avoid common points of failure.

In order to build the chains, first classify all machines as either Taken or Reserved (initially all machines will be Reserved). Further, the machines can be put into different failure zones, the number Z of zones not necessarily being the same as the number of chains C nor the size of a chain K. We can create a sorted list for every zone, delivering machines in descending size of available store (number of partitions hosted). We also total the size of all resources and reserve some fraction for future use. The size goal for each chain will then be GoalTotalPartitions/C. We can then cycle through the zones for each chain, taking a machine from the zone to add to the chain, until the size goal is met.

Large machines are expected to be preferred due to reducing the number of entities, and maximizing the choice of servers with little competing use of SSD. However, this policy could be changed in future. In general we just want some “best choice” function which selects the next machine based on size, other tenants, length of lease, quality of the SSD, etc.

Later when a machine is lost from a chain we can look at which zones are represented in the chain, choose a zone which is contributing the smallest number of partitions, and then add the best reserved machine available out of that zone. If the result does not add up to the target size for partition count within a chain, then we can repeat the process with the next best chain and server.

It seems a reasonable simplification to assert that the primary, secondary, and tertiary storage all us the same definition of chains. So, we build the chains based upon the SSD, and terms of service for the hosting of EB partitions is that the server must supply HDD double the size of the SSD.

Chains must be universally agreed. So, this process of assigning servers to chains is happening by consensus in the EB Managers, and then is diffused out to update all the partition sets in Brokers.

## Mapping partitions within the Chains

Each key maps to a partition using some stable hash function, and each partition maps to a chain by a stable hash function (which might be as simple as modulo C). However, there may be hundreds of partitions per chain. Initially this assignment is done by creating a list of all the physical partitions in each chain, ordered in some way which spreads the load fairly among different sized machines, and then the logical partitions each get mapped in turn to a physical partition for the primary, secondary, and tertiary use. The three uses avoid machines from the same zone.

These choices are what we need to put into the partition map, and they can be recorded with a small number of bits. For example if K is less than 256 machines then we can encode the primary map with one byte. The entire map of 100M partitions in a maximal implementation will then be 100 MB. The size needed to map 50 PB of data in one of today’s DCs is closer to 5 MB. The client Broker will generally be able to keep secondary and tertiary maps on SSD, just allowing enough DRAM to cache recent changes to the map or recently failing over partitions, which at any given moment should be a small memory commitment (probably a small unordered\_map<> or similar).

To summarize, the process of locating the correct server is:

1. Hash the Key down to a partition number
2. Divide the partition number down to a chain
3. Use the log2(K) bits of map[partition] to select the server within the chain

In case of failure of the server, the secondaryMap[partition] or tertiaryMap[partition] will be used to find the secondary and tertiary. In practice those maps can be combined with an entry double log2(K) bits wide (probably 2 bytes), so we only need one lookup on local SSD to be prepared to talk to either secondary or tertiary, and also to update them in one place if we do decide to failover to a new machine.

## Failures

In case of a machine failure those partitions are lost and have to be replaced by secondary and tertiary copies, and a new copy inserted. The replacement partitions must map within the same Chain. This is because in order to be economical on the size of the partition map (1 byte per entry) we have no flexibility in mapping the partition to a different chain. The map is a stable function.

Since we want a fast parallel failover this also implies that chains should be big. If we want to allow one machine to have 100 partitions and the partitions to all failover to different other machines, then the chains should have at least 100 machines in them.

As the number of machines in a chain increases it will be easier for us to balance them nicely. 128 machines with maybe 5,000 partitions are going to be easier to balance than 16 machines with 800 partitions. The main downside is the map is bigger than was earlier estimated. Earlier claims expected the map to be just 4 bits per partition instead of 1 byte. Compressing the map can be revisited later.

When a failure occurs each partition will go through some phases. The first phase will be to redirect both reads and writes to the secondary while leaving the primary map in a probationary state. We might reserve one map value to indicate this and then keep an unordered\_map<> of re-routing information. We also notify the EB Managers of our suspicion. Eventually either the primary comes back online and the probation is reversed, or the EB Managers decide to replace it. Upon replacement we promote the secondary and tertiary, and the EB Manager will supply a value for the new tertiary.

There is a similar state machine for the secondary to go through probation where we rely entirely on the tertiary, and if we can’t talk to the tertiary then our operations fail back to the application.

## Growth

We want to ensure there is no upper limit to the number of partitions, while keeping our resources in proportion. Today we might need only 4 or 8 million partitions to cover the 50 PB or so of SSD store we hope to recover in each data center. In 5 years 128 million partitions might be needed – but the servers should have much more capacity so a broker map that size may be fine. The problem is not really the size. The problem is allowing us to grow our size over time.

We do not make any plans for shrinking the map. In principle the process described below can be reversed but it seems unlikely the code for that would ever be used. It can be written later if needed.

In order to grow the plan is, in broad outline:

1. Double the number of partitions and chains. The partitions and chains are identical twins.
2. Maintain the twinning until all clients and servers acknowledge the change. Begin phase 3.
3. Now when a machine fails, replace it separately in each of the two chains. This is controlled by consensus in the EB Manager. The clients will have been using the doubled maps so will be oblivious to the fact that the twins drift apart, since actually they are not using them as twins.
4. In an Imp-style process go through the machines which have not actually failed and issue failure for them in just one of the chains.

What is happening here is that, unlike the partition map, the Chains were constructed as lists and we can tolerate multiple references for a while, so long as we begin the split early enough to be confident that the partitions will not overflow during the period they remain targeted by both twins. Indeed the Imp process for forking them can be driven by searching for machines with partitions most at risk of filling up. However, the whole sequence should be ok to play out over weeks since there is no logical error in the map. We can also time the Imp to not run at peak times. This allows us to reduce the stress on the data center due to migrating data. Half of the data will eventually be moved.

# Partitions at the Servers

Partitions at the server should be much less complicated.

There are of course still up to 100M partitions in the system – but each server is only handling a few hundred of them. We can hash the key values down to whatever the current cardinality of partitions is, and then use an unordered\_map<> to correlate those to our local EbPartition. The EB Managers will tell the servers which partitions they are serving.

The main complication is likely to be at times of growth when there are seconds or even minutes before all servers and clients are on the newly twinned partition counts. In order to handle that I think we will need some sort of versioning in our Datagrams. We twin the maps, and maintain the originals and the twins all the same. During that period clients may be sending requests of either old or new version, and the server should use the version to decide which set of maps to use, but either are valid. The manager will, during this phase, only send failover and other maintenance messages based on the old partition count so we only need unidirectional logic to keep everything matched.

After the managers have observed all machines are switched to the new partition count, and a prudent delay for straggling events, it should send out an end end of transition event. The servers can discard the old maps and fail any datagrams from the prior versions, since none should be arriving.

The Imp process of removing servers from the client chains should be matched by sending cancellations to the servers, informing them of partitions they are no longer serving. These will actually be fairly drastic since typically half of the partitions will vanish leaving that server fairly underused. It will be expected to get more use in coming days as it becomes preferred for collecting failovers from other machines, since the failover process should prefer underused machines. These cancellations can be scheduled over a period of time and avoiding peak activity times.

# EB Manager

EB Managers are a group of machine forming a highly reliable cluster using either RSL or ZooKeeper. They are in charge of managing the partition configuration for the whole store.

## Matching servers to Chains

The EB Managers will observe a variety of conditions which change the mapping of servers to Chains. In general you will observe this process depends upon maintaining a reserve of unused machines, a reserve of unused physical partition files within the set of machines assigned to a chain, and the process of Imping as well as normal failures driving a diffusion of changes always chasing the ideal steady state.

Most changes are expected to be gradual. Major outages will be the exception but even there the system is going to run at some throttled speed. EB is built for 3-9s availability and for datasets which are tolerant of occasional losses. It may be improved for higher service later, after gaining operational experience at this more achievable level which we believe still has much practical use.

### Environments newly enable hosting of EB partitions

This is the starting condition and will also occur from time to time. Multiple servers become available and each has some quota of partitions it may support. We expect this quota to be uniform, same count for every machine within a partition. The administrative dashboard for EB should have investigated available space and validated that all machines in the environment have the required free space.

Once the tools nominate an environment to the EB Manager, the machines in the environment are enumerated and added to the Reserved pool, one by one but as fast as the EB Manager can accept them. We move machines from Reserved into active chains at a measured rate. When existing servers fail (including Imping) we generally respond by adding the largest available Reserved machine to that chain, unless the chain already has more partitions than current goals. We also start picking undersized chains and adding the largest reserved servers when our overall capacity is below the current goal.

Adding a whole environment does not directly increase our goal for overall capacity. We monitor usage. There will be a formula which provides both some absolute excess capacity, and some proportional excess capacity, beyond the amount actually in use. In general we can hope that environments volunteer more space than we actually need. This is going to result in us not making use of machines which provide the smallest quotas. Even if small capacity machines are in use they may fall out of use when they fail or are Imped, if some larger capacity machines happen to be in reserve. We should do this to keep the system honest and the accounting for storage needs transparent.

We will start the EB Service on each machine as it becomes ready for service and pre-emptively reserve the partitions. This process might be kicked off for a number of the largest machines in Reserve just to lower the average time to make them active if we need them. We can call these pre-Active. But, most of the reserve machines will have no partitions allocated. They will, however, be running the EB Server in a passive mode which keeps a watchdog alive to the EB Manager and that watchdog includes monitoring available space so even if the environment is largely in reserve we will have alerts when its quota is unsafe.

Once the machines become active they will start being assigned partitions as a result of machine failures and Imping. It may take some time until all partitions are in use.

### Environments increase the partitions allowed

If an environment increases partition quota then we can pre-allocate more files in the active and pre-active machines from that partition. They will not immediately be used. The partition maps for the chains will migrate to use these as a result of failovers and Imping.

### New individual servers added

Servers may be added individually to environments as they come back from failed states including Imp, transient fault, or repair. Such machines are added to the reserve pool and will become pre-active then active as with any other machines in the reserves.

They may be added in sets if the environment is enlarged. The process is really the same for one or for many. We would assume the EB Manager is capable of accepting and processing a new machine in less than 1 second, allowing for at least 80,000 additions per day. The main reason to design for at least that rate is to allow rapid recovery from major site failures.

### Existing individual servers fail

Failing machines need to be removed from their chain and, if the chain is below goal, a replacement will be assigned from reserves. When the machine is repaired (which includes recovery from Imp or self-test) it will NOT be automatically returned to the same chain. It becomes a new individual server added to the reserves in the usual way.

### Services decrease the partitions allowed

The free space ratios on servers should be monitored and alerts generated if some of the machines in an environment appear unable to support the quota. There are no current plans to allow quotas to adjust on a per-machine basis. The assumption is that environments are uniform. We might have some automation on reducing quotas across the whole environment if machines start to fill up and no human seems to be responding to alerts.

When the services decrease they do so over a period of time, we would expect not to change more than 5% per day. This will be implemented as a steady stream of single server eliminations. The cuts will be evenly distributed across all the machines in the environment and then a schedule of Imping will be used to delete partitions at a moderate rate, avoiding peak times. The capacity will be replaced from the normal process of nominating reserves when chains fall below goals for total and for spare (unused) partition files, and mapping logical partitions to unused physical files on remaining machines within the chains.

### Environments terminate EB hosting

This is the extreme case of reducing allowed partitions. It works the same and will be expected to follow the same limit of 5% change per day.

Issues of the dashboard monitoring projected overall shortfalls and possible forced reductions in service for some tenants are not going to be handled in this first version. We will probably aim for simple alerts, and leave it up to manual processes to decide which tenants must reduce if major reductions happen.

# Interaction with PacificA

PacificA is the protocol chosen to manage replication and availability.

The “servers” in PacificA for our purposes are not the whole server machines, but rather the individual partitions on each server. The state machines, the leases, the communication between primary and secondary, all of these run independently for each partition. A server *machine* could have three hundred or more independent PacificA servers running to track all the primary, secondary, and tertiary partitions.

The EB Manager provides policy and recording. The partitions provide implementation and execution. Thus the partitions maintain replication and leasing. The partitions inform the manager about changes in state such as failed leases. The Managers also get state information from client brokers (“partition X seems not to be responding”) and will have some ability to directly ping a server or a partition. As a result, the Managers maintain an overall picture of the status of machines and partitions. This map is not synchronized perfectly to the partitions, rather it will in general lag the actual state of the partitions but be eventually consistent.

Certain actions of the partitions should obtain advice from the managers. For example when a partition fails its partners need a replacement. They should not elect the replacement themselves because they have no information about the state of servers or partitions outside of their own small redundancy set. A good choice of replacement will need information about independent fault domains, machine loads, ordering alternatives to choose the best. The Managers can do that. Now, it is possible they make an obsolete choice. A major mistake would be selecting a partition which has failed but the Manager does not yet know. That rare mistake should self-correct because the partitions will soon discover it is unresponsive and will retry with the Managers for another selection. A minor mistake would be choosing a machine which is not optimal, because the server is out of date. That does not matter. Many machine groups will have become non-optimal, the design only requires that we keep making good decisions which over a process of slow churn will keep the configuration close to optimal – at least, far away from bad configurations.

The view which client brokers have is provided via the managers, too. This means it may be slightly delayed, but the managers are in a better position to aggregate the changes and ensure that the partition map is distributed from snapshots which are self consistent.

# Gossip Protocol

Used to propagate partition map and chain table to all EB Library instances.

There may be a huge number of clients with brokers, and the brokers all need to maintain a recent map. The map information is controlled by the Managers but if the managers need to send updates to 100,000 brokers this will be a problem. We want to keep the delay under a second for map changes to reach all client brokers. The shorter this time, the better availability will be.

In order to solve this we will use a semi-optimized Gossip network. In the original Gossip each machine chooses N neighbors at random. Incoming changes are merged, with an eventually consistent mechanism, and then any changes which were not already known are packaged into new messages and sent onwards to the neighbors. By diffusion eventually all machines will be reached.

This mechanism can be improved by choosing better neighbors. We want to avoid keeping links to extinct neighbors. We want to avoid having too many neighbors which share the same failure zone. And we want to avoid having neighbors which are already closely connected. All of these avoidances have the effect of increasing robustness and of lowering the average hops to diffuse changes.

The way this is done is fairly simple, using the Managers. The Managers keep two lists for every broker:

1. The outgoing neighbors
2. The incoming neighbors

For each machine we build each list inductively. The initial state for a list is a random choice of broker, constrained to be (i) not in the other list, and (ii) not in the same fault zone as the current broker. Then the list is filled out by randomly picking brokers which (i) are not on either list and (ii) rotate through fault zones and (iii) minimally connect to the existing lists.

Step (iii) is the tricky one. We cannot really minimize globally, that would be very expensive. However, simulation of gossip networks shows they generally work well as long as you are not unlucky. To avoid being unlucky all we need is to generate several alternatives, and always choose the best alternative. Since we are dealing with large numbers and consistently biased towards a good choice, our networks will be robust and efficient. In practice we should be fine with generating 4 or 5 alternatives, evaluating their score for (iii), and keeping the best one.

The managers can add gossip lists in batches when more than one broker is ready to add. This is mostly important when waking up whole environments. We should rotate through the machines adding one in and one out neighbor at a time to each broker so that the networks fill up in a balanced way. We can also keep an elastic definition of a full list, so for example the definition of a full list might be 10 neighbors but when brokers are down to 9 or 8 neighbors we just put them into a spare pool until a new machine comes along, or we get a broker that falls to 7 or below. That way we have plenty of choices for good neighbors for new machines. We can also Imp the lists, going around to brokers with 10 connections and disconnecting 1 or 2 so that the broker goes into the pool and forms rebalanced connections. We can also have brokers elect to reconnect if they detect that they are getting changes which have had to pass through more than some upper limit of hops.

The manager should choose a different random set of starting brokers for each broadcast.

## Eventual Consistency

As the map is spread across the gossip network we need eventual consistency. As is common, a version based system is used where the Manager will stamp each change with a (cyclic) monotonic increasing version number. Upon arrival the changes are compared to the existing information. Values which are already known are ignored. Values which are different are compared for version, and the latest version wins. If the incoming change is different and the latest then it is applied to the local state, and added to a list of outgoing changes. After a short time, which allows aggregation of multiple incoming messages, all pending outgoing changes are gathered into a message and sent to the outgoing neighbors.

The diffusion speed can be improved by modifying the latency depending on how many incoming novelties were seen. This is because if the changes were novel we can infer that most neighbors will not yet have seen the changes so there is higher value in getting out the news. This can also be approximated by using a threshold for sending changes immediately when more than some number are waiting, which balances the need for efficient use of messages (multiple changes per message will reduce overheads) against the desire for rapid diffusion.

This fairly classic mechanism allows random diffusion of updates while converging consistently and avoiding redundant communication. Changes may arrive multiple times but any given change will leave exactly once. The network has a level of inefficiency and redundancy which increases with the number of neighbors but as this information is expected to be small this is a fair tradeoff to get the benefit of a diffusion network in reducing the latency and eliminating hotspots of just a small number of machines trying to send a large number of messages.

It may be noticed that when we have millions of partitions, we do not want to have to keep version numbers for all of them. We already worked hard to compress to 8 bits per partition and hope in future to maybe get down to 4, so adding version information would be very expensive. Fortunately, this is not possible. Firstly, if the incoming change is not different than the existing value then we do not care about the version. There is no change, there is no action and we have consistency. For those values that disagree the only question is, do we already have a later value? All that requires is keeping a short list of the most recently seen changes (including ones which agreed with existing values, since they may establish a more recent version). As the incoming change is likely to have fairly recent versions it is just a short scan to establish if it is not the most recent. We do not need to keep an infinite list. We can discard changes that came in more than some time ago. Since we aim to have complete diffusion in less than a second, we should be absolutely safe with a 10 second history, and ridiculously safe with 100. This is expected to be a negligible memory footprint and, since only exceptional cases would search more than a fraction of a second of history, a negligible performance cost even when searched linearly.

Note that we can assert that we should never see a new message arrive which has a version earlier than the list. That would indicate something has gone very wrong with the diffusion mechanism and enable us to monitor that we have in fact got the correct, safe length of history. Indeed we can log a histogram of actual search lengths as part of the real time monitoring of service health.

The Gossip mechanism will be used for broadcast of any information the Managers need to communicate to all Brokers. However, there may be other activities such as initialization and the reporting of faults where individual Brokers may directly exchange messages with the Managers.

# Tracking Storage Owners

Storage owners are the services which sign up to use the Exabyte Store. Only registered services can be writers and we will issue 64 bit numbers as IDs which authorize writing and, optionally, reading. We need to track usage and quotas for a service. A service has a uniform behavior in terms of storage lifetime, which must be declared along with the quota. Typically services will have several current IDs which we replace over time to make it harder for hacking to have permanent effect.

More details needed.

<end of work in progress>

# References

**Lin, Wei, et al. 2008.** *PacificA: Replication in Log-Based Distributed Storage Systems.* s.l. : Microsoft Research, 2008.