Hybrid Logical Clocks for HBase and Phoenix

Enis Soztutar
Hortonworks
July 2015

Problem statement - HBase

HBase and Phoenix uses systems physical clock (PT) to give timestamps to events (read and writes). This works mostly when the system clock is strictly monotonically increasing and there is no cross-dependency between servers clocks. However we know at least these cases are breaking the correctness guarantees

- Leap seconds and other non-monotonic updates to system clock (misconfigured NTP, virtual OS failing to keep up with the underlying hosts physical clock, etc) can cause the PT to go back.
- On windows systems the precision of the PT is ~16 ms (meaning the system clock does not update for 16 ms, causing all the events to have the same timestamp).
- Read-modify-write operations (like Increment, Append, checkAndPut, etc) are more susceptible to PT going back in time.
- HBase orders cells by type first, then seqid, which means that a DELETE which happens before (hb) a PUT will ecplise the PUT in the same millisecond.
- No notion of global snapshot point across physical clocks making SI / snapshot / backup susceptible to clock skew across the cluster

Problem statement - Phoenix

- Phoenix uses regular system clock for (1) cache update / invalidation for metadata operations, (2) do writes with server time from the system clock of region server(s) that the writes are scheduled and (3) do reads with the system clock of the metadata server
- Phoenix implements READ_COMMITTED through system clocks. The read / write timestamps being acquired from different servers make Phoenix extremely vulnerable to clock skew between servers, and affects correctness
- The cache lookup / invalidation is also vulnerable to clock skew in case the metadata region moves or leap seconds or other non-monotonic updates to the system clock

Related Jiras

General discussion:
HBASE-8927 Use nano time instead of mili time everywhere
HBASE-6833 [WINDOWS] Java Milisecond precision on Windows

Writes getting eclipsed:

HBASE-12449 Use the max timestamp of current or old cell's timestamp in HRegion.append()
HBASE-14054 Acknowledged writes may get lost if regionserver clock is set backwards
HBASE-6832 Tests should use explicit timestamp for Puts, and not rely on implicit RS timing

Region assignment / updates get eclipsed:
HBASE-11536 Puts of region location to Meta may be out of order which causes inconsistent of region location
HBASE-13709 Updates to meta table server columns may be eclipsed
HBASE-13875 Clock skew between master and region server may render restored region without server address
HBASE-13938 Deletes done during the region merge transaction may get eclipsed

Introduction

Hybrid Logical Clocks (HLC) have been proposed recently in [1] which is an implementation of an hybrid physical clock + a logical clock. HLC is best of both worlds where it keeps *causality relationship* similar to logical clocks, but still is compatible with NTP based physical system clock. HLC can be represented in 64bits.

You can skip the rest of this section if you have read the paper (which is highly recommended). Below are excepts from the paper:

Logical clock (LC). LC was proposed in 1978 by Lamport as a way of timestamping and ordering events in a distributed system. LC is divorced from physical time (e.g., NTP clocks): the nodes do not have access to clocks, there is no bound on message delay and on the speed/rate of processing of nodes. The causality relationship captured, called happened-before (hb), is defined based on passing of information, rather than passing of time. While being beneficial for the theory of distributed systems, LC is impractical for today's distributed systems: 1) Using LC, it is not possible to query events in relation to physical time. 2) For capturing hb, LC assumes that all communication occurs in the present system and there are no backchannels. This is obsolete for today's integrated, loosely-coupled system of systems.

In 1988, the vector clock (VC) [7, 19] was proposed to maintain a vectorized version of LC. VC maintains a vector at each node which tracks the knowledge this node has about the logical clocks of other nodes. While LC finds one consistent snapshot (that with same LC values at all nodes involved), VC finds all possible consistent snapshots, which is useful for debugging applications.

Physical Time (PT). PT leverages on physical clocks at nodes that are synchronized using the Network Time Protocol (NTP) [20]. Since perfect clock synchronization is infeasible for a distributed system, there are uncertainty intervals associated with PT. While PT avoids the disadvantages of LC by using physical time for timestamping, it introduces new disadvantages: 1) When the uncertainty intervals are overlapping, PT cannot order events. NTP can usually

maintain time to within tens of milliseconds over the public Internet, and can achieve one millisecond accuracy in local area networks under ideal conditions, however, asymmetric routes and network congestion can occasionally cause errors of 100 ms or more. 2) PT has several kinks such as leap seconds [13, 14] and non-monotonic updates to POSIX time [8] which may cause the timestamps to go backwards.

TrueTime (TT). TrueTime is proposed recently by Google for developing Spanner [2], a multiversion distributed database. TT relies on a well engineered tight clock synchronization available at all nodes thanks to GPS clocks and atomic clocks made available at each cluster. While TT avoids some of the disadvantages of LC/VC/PT, it introduces new disadvantages: 1) TT requires special hardware and a custom-build tight clock synchronization protocol, which is infeasible for many systems (e.g., using leased nodes from public cloud providers). 2) If TT is used for ordering events that respect causality then it is essential that if e hb f then tt.e < tt.f. Since TT is purely based on clock synchronization of physical clocks, to satisfy this constraint, Spanner delays event f when necessary. Such delays and reduced concurrency are prohibitive especially under looser clock synchronization.

HybridTime (HT). HT, which combines VC and PT clocks, was proposed for solving the stabilizing causal deterministic merge problem [10]. HT maintains a VC at each node which includes knowledge this node has about the PT clocks of other nodes. HT exploits the clock synchronization assumption of PT clocks to trim entries from VC and reduces the overhead of causality tracking. In practice the size of HT at a node would only depend on the number of nodes that communicated with that node within the last time, where denotes the clock synchronization uncertainty.

HLC: Hybrid Logical Clocks

HLC is a way to capture causality relationships while maintaining the clock value to be close to the system clock (NTP). Each node has a clock, which have two components the physical component (PT) and a logical component (LT). Each node can also access to its own system time (ST). In plain english, the algorithm for HLC is quite simple. This pair <PT, LT> sits on top of the system clock, and is the HLC. The HLC in some sense keeps updated with all the known values from clocks of the other cluster nodes and is always monotonically increasing.

- When creating a local event, get the system physical time (ST). If system time is greater than PT, then set PT as ST, and reset the LT.
- Otherwise, ST went back in time, or is in the same precision (for example in the same millisecond). Then increment LT by 1.
- If an event is received with timestamp <msgPT, msgLT>, then check ST and do:
    - if ST is greater than both PT and messages PT, then set PT to be ST and reset LT. This means that our ST has advanced.
    - if remote messages PT is greater than PT, then set PT = msgPT and LT to be 1 more than msgLT.

    - if local PT is greater than remote messages PT, then increment local LT by 1
    - else messagePT should be equal to local PT. Set the local LT to be 1 more than max of PT and msgPT.

HLC as presented in the paper has very desirable properties that seem like a perfect fit for HBase and Phoenix:

- HLC refines both the physical clock (similar to PT and TT) and the logical clock (similar to LC). HLC maintains its logical clock to be always close to the NTP clock, and hence, HLC can be used in lieu of physical/NTP clock in several applications such as snapshot reads in distributed key value stores and databases.
- HLC preserves the property of logical clocks (e hb f ⇒ hlc.e < hlc.f) and as such HLC can identify and return consistent global snapshots without needing to wait out clock synchronization uncertainties and without needing prior coordination, in a posteriori fashion
- Timestamp assigned by HLC is “close to” physical time, ie. |l.e − pt.e| is bounded.
- HLC is backwards compatible with NTP, and fits in the 64 bits NTP timestamp format
- HLC works as a superposition on the NTP protocol (i.e., HLC only reads the physical clocks and does not update them) so HLC can run alongside applications using NTP without any interference.
- HLC provides masking tolerance to common NTP problems (including nonmonotonous time updates) and can make progress and capture causality information even when time synchronization has degraded
- HLC is also self-stabilizing fault-tolerant and is resilient to arbitrary corruptions of the clock variables
- HLC works for a peer-to-peer node setup across WAN deployment, and allows nodes to use different NTP servers
- Finally, while HLC utilizes NTP for synchronization, it does not depend on it. In particular, even when physical clocks utilize any ad hoc clock synchronization algorithm [17], 6 HLC can be superposed on top of such a service, so can also be used in ad hoc networks.

HBase implementation notes

- HLC can be modeled as clock per node or a clock per region. However, for possibly ordering events across regions, clock per node is better.
- We have three types of events for a given node. Local events, send events and receive events. All actions that end up depending on EnvironmentEdge.currentTime() is a local event. Receive events are from RPC and through other indirect communication between nodes (for example region move)
- That means all requests (put, get, scan) will be a local event. We can also extend this for other events such as flush, compaction, log roll, region opening, region closing, etc.
- For HBase, usually, the nodes do not send each other messages except for master <-> RS communication, and region movements. For region movements, it is always: region closed by RS1 *happens before* region opened by RS2. Thus we will model the region

close time (or the maximum timestamp of any edit in the region) as a send event from RS1 to RS2. This ensures that the region cannot have any writes with a smaller timestamp than the previous server assigned.

- We should model all “happens before” events to go through the HLC clock by properly using local, send and receive events. Possible happens before events are like:
    - update_1 happens before update_2 => HLC(update_1) < HLC(update_2)
    - update_1 happens before read_1 => HLC(update_1) < HLC(read_1)
    - region close on RS1 happens before region open on RS2 => HLC(region_close) < HLC(region_open)
    - master calls region assign happens before meta update by region server => HLC(assign) < HLC(meta_update)
- We want to keep HLC representation to be 64 bits to be BC with cell timestamps. Ideal representation of 64bits physical + 64bits logical gives a lot of flexibility, and is nearly impossible to overflow, but we cannot store 128 bits per cell without major surgery. Plus 64bits seems like a good tradeoff sufficient precision (millis) and max concurrency within millis.
- HLC timestamps will be different (and not interoperable) with currently used System.currentTimeMillis() based timestamps. Existing tables should not be converted to HLC timestamps, but new tables should default to using HLC based timestamping. Wall clock based features like TTL should work with both implementations of ‘wall time’.
- Extreme care should be taken to not interpret timestamps of one type (HLC) versus the previous one (currentTimeMillis).
- HLC is connected to the actual system PT and the divergence is bounded. We can keep the TTL and similar features with HLC.

Java time resolution

There seems to be no easy way to get ns or micros precision clock in jdk8 via gettimeofday() or clock_gettime(). Possible candidates are System.currentTimeMillis(), System.nanoTime(), Instant, Perf.highResCounter().

./src/os/linux/vm/os_linux.cpp in http://hg.openjdk.java.net/jdk8/jdk8/hotspot/

```
jlong os::javaTimeMillis() {
  timeval time;
  int status = gettimeofday(&time, NULL);
  assert(status != -1, "linux error");
  return jlong(time.tv_sec) * 1000  +  jlong(time.tv_usec / 1000);
}

jlong os::javaTimeNanos() {
  if (Linux::supports_monotonic_clock()) {
    struct timespec tp;
    int status = Linux::clock_gettime(CLOCK_MONOTONIC, &tp);
            assert(status == 0, "gettime error");
```

```
            jlong result = jlong(tp.tv_sec) * (1000 * 1000 * 1000) + jlong(tp.tv_nsec);
    return result;
  } else {
            timeval time;
    int status = gettimeofday(&time, NULL);
            assert(status != -1, "linux error");
            jlong usecs = jlong(time.tv_sec) * (1000 * 1000) + jlong(time.tv_usec);
    return 1000 * usecs;
  }
}
```

the System.nanoTime() uses CLOCK_MONOTONIC [3] on Linux and BSD. It is not immediately clear whether CLOCK_MONOTONIC can be combined with the millis resolution clock obtained via currentTimeMillis().

There seems to be a couple of options going forward:

1. Continue using System.currentTimeMillis() regardless option 1 or option 2 as below is chosen (more precise clocks can be added later in option 1).
2. Use gettimeofday() through JNI or JNA. JNI requires shipping native code, versus JNA option seems that performance is not clear.
3. Find a way to reliably combine System.currentTimeMillis() + System.nanoTime() across different hardware threads.

Representing HLC in 64 bits

We would like to represent PT and LT combined in a 64bit long. This will be compatible with the cell's timestamp and all other timestamp usage in HBase (like TTL, WAL, HFile timestamps, RS timestamps, etc). The representation of the combined timestamp MUST be comparable using regular Java long comparison. This also means that without changing the HBase's default CellComparator, we will only have 63bits as the MSB is for the sign.

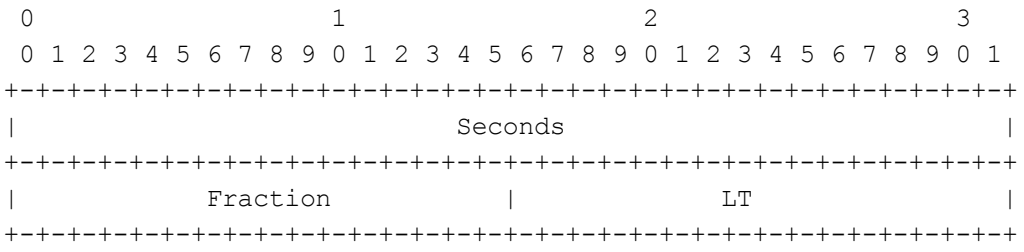
NTP represents the timestamp as an 64bit value, where the first 32 bits are for seconds, and the next 32bits are used to represent the fraction of a second. Unlike unix epoch, NTP starts at 1900, and the first 32bit will roll over at year 2036. The highest order 32bits can represent 136 years with seconds as resolution.

The fractional part of NTP time can represent a precision of 1/2^32 ~= 232 picoseconds.

Representing a PT and a LT by combining two numbers into one requires a choice between different tradeoffs. We cover a couple of options below.

Option 1: As in HLC paper

HLC paper suggests using NTP time, and rounding the 64bit NTP number up to the 48bits and using the remaining 16 bits for representing LT. This gives a theoretical resolution of 1 / 2^16 ~= 15 microsecond. 16bits used for LT means that 2^16 ~= 65536 can be used.



Using this scheme requires either changing the year base from 1900 to be 1970 to deal with Java long comparison, or fixing all timestamp comparisons in HBase to be based on unassigned long comparison. The former is acceptable since current implementation is already epoch based. In this case years up to 1970 + (136/2) ~= 2038 can be represented.

Option 2: Shifting Java millis

Option 2 involves keeping the 1ms resolution from Java, and using higher order N bits to represent the epoch, and use lower order 64-N bits to represent LT.

Bits for PT	Bits for LT	MILLIS / YEARS	MICROS / YEARS	MAX LT
40	24	34	0	16,777,216
42	22	139	0	4,194,304
44	20	557	0	1,048,576
46	18	2231	2	262,144
48	16	8925	8	65,536
50	14	35702	35	16,384

In this scheme, representing PT with 1 nanosecond precision requires at least 62 bits for PT, which means 146 Years can be represented. This leaves 2 bits for representing LTs.

As can be seen, the 64bit representation of PT + LT is a tradeoff between (a) the precision, (b) maximum year that can be represented, and (c) maximum concurrency for LT within the same precision.

If we assume that we would not be able to achieve <1ms precision from Java for linux deployments without writing a JNI wrapper, or find a way to combine ns+ms from system, than Option 2 with 42bits-per-PT, or 44-bits-per-PT with ms precision seems like the best option.

Otherwise if we want to be flexible and we want to get micros precision, we can go with the Option 1 approach. Notice that with option 1, we can start with ms precision and if we can find a way to do better resolution, then we can compatibly change the implementation without affecting currently written data.

Dealing with LC overflow

We have to deal with possible overflow coming from LC counter. This means that if LC overflows we have to take an action to make sure that event ordering is not affected. Only way to possibly deal with LC overflow seems to be deferring the event to be executed in the next PT increment (by making sure the systems PT advanced and LT is reset). This limits the throughput with 2^bits_for_lt per node per time resolution. Practically it is unlikely to overflow LC with 16 bits, and at least 1 ms resolution which enables 65K events per ms.

Clock drift protection

HLC proposes to use an upper bound on accepting messages which have a pt larger than allowed, Δ. Δ can be a constant factor of E (crudely two times the NTP offset value). HBase already has a bigger upper bound on the max clock drift allowed (30000 ms by default) which can be used for ignoring / rejecting messages.

HBase implementation action items

1. Implement base HLC algorithm, including send/receive, 64bit encodings, java system clock workarounds (JNI, etc) and unit tests
2. Micro-benchmark different HLC implementations and 64 encodings to find the optimal one
3. Introduce a new Table property for using HLC for the cell's timestamps (and other timestamps) for the table. Enabled by default
4. Abstract usages of EEM.currentTime() so that either the deprecated one or HLC can be used (this is mainly in HRegion)

5. Region open updates HLC with the highest timestamp from the region's files or close region event.
6. Add a user API for interpreting the timestamp from HLC. Converting Cell.getTimestamp() to epoch based date.
7. Region assignment uses HLC for all the operations (open region rpc, region open timestamp, meta update, etc)
8. Make Master <-> RS RPCs use HLC and every heartbeat / region open, type of message exchanges updates HLC. Note the HLC time can be used to globally order region events for logging / debugging if events are ordered with "happens before".
9. TTL feature should be made compatible.
10. Replayed edits through replication should update HLC of the node so that master-master setup makes sure that timestamps don't go back.
11. HLC usage can be extended to other places where there is explicit causality (master procedures, etc).

Phoenix implementation notes

- Cache updates for the client and metadata server can use HLC so that the clock is guaranteed to be monotonically increasing
- All writes can use the metadata servers HLC to send with the write, so that the write commit ts from the regionserver is guaranteed to be > metadata servers HLC at the time of the statement begin.
- Reads will use the HLC timestamp as an upper bound.
- Transactions for Phoenix can use the HLC instead of a global ts oracle and still keep the relation to real time.

Phoenix implementation action items

TODO

References

[1]http://www.cse.buffalo.edu/tech-reports/2014-04.pdf
[2] https://tools.ietf.org/html/rfc5905
[3] http://man7.org/linux/man-pages/man2/clock_getres.2.html