# Java Big Memory: Replication for Resiliency and Load Balancing of Milestoned Data

Mohammad Rezaei

June 2014

# Agenda

- MasterCacheService
- Replication data organization
- Replication data stream
- Master sync algorithm
- Replica sync algorithm
- Removing data from the master cache
- Restarting Master

- What is cache replication?
- Motivation
- When is replication appropriate?
- C-heap data storage and Strings
- How does replication work?
- Milestone Consistency
- Performance numbers

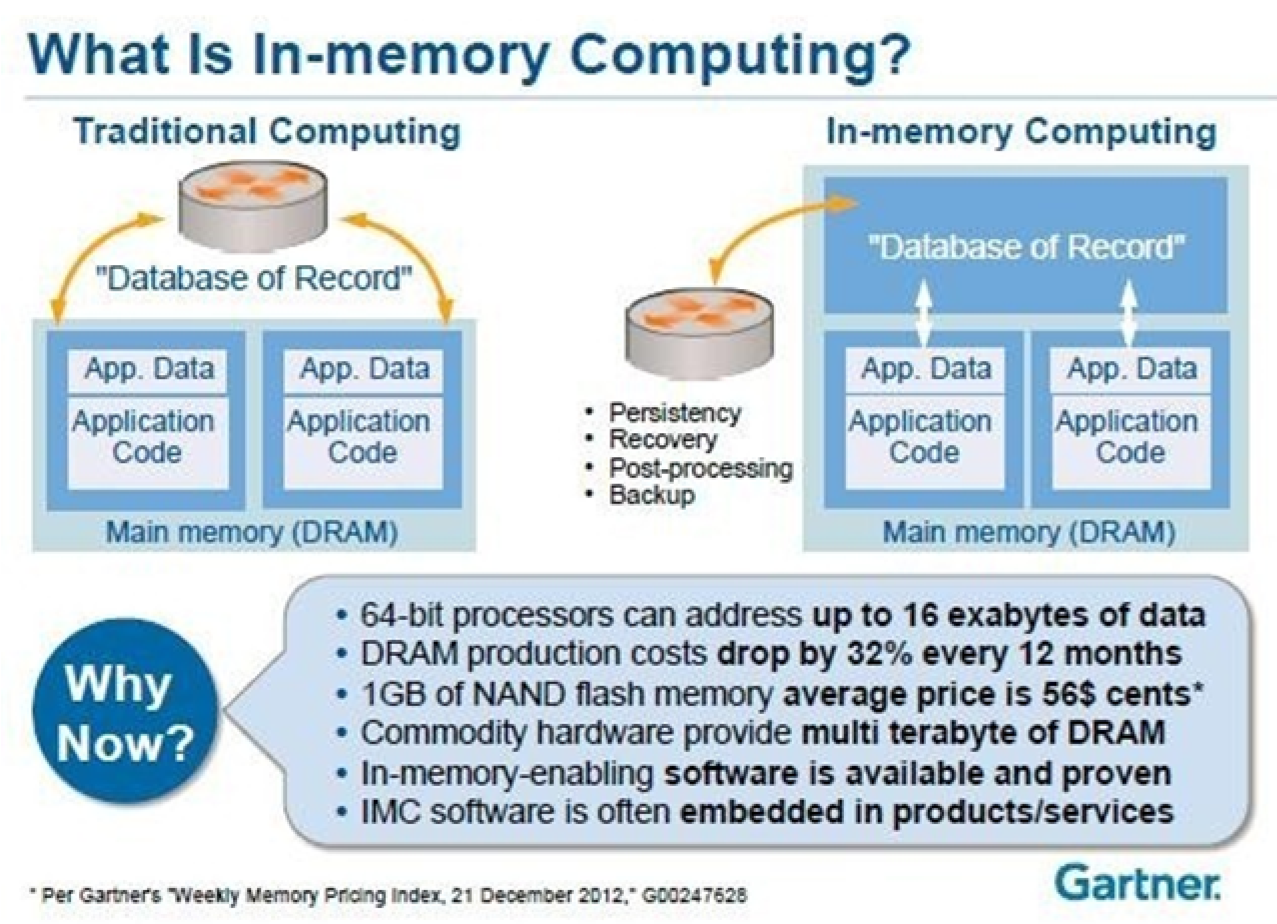
# What is replication?

- No data is missing
- When master is updated, the replica can catch up (efficiently).
- The application knows the status of the replication so it can construct correct queries (it must not query data that the replica doesn’t have).
- Cache replication involves copying data from a *master* cache to one or more *replica* caches.
- Replication has to correctly copy the data so that:
- Each Reladomo class has its own cache. Replication is performed at the same (class) level.
- A replica typically has a single master, but it may have multiple masters (for different domains/classes).
- A replica can have non-replicated classes.
- A replicated class is read-only on the replica.

# Why an in-memory store?

- "Gartner Says In-Memory Computing Is Racing Towards Mainstream Adoption"
- http://www.gartner.com/newsroom/id/2405315
- http://www.slideshare.net/SAP_Nederland/the-next-generation-architecture-inmemory-computing-massimo-pezzini

# Why an in-memory store? (Continued)



## What Is In-memory Computing?

- Persistency
- Recovery
- Post-processing
- Backup

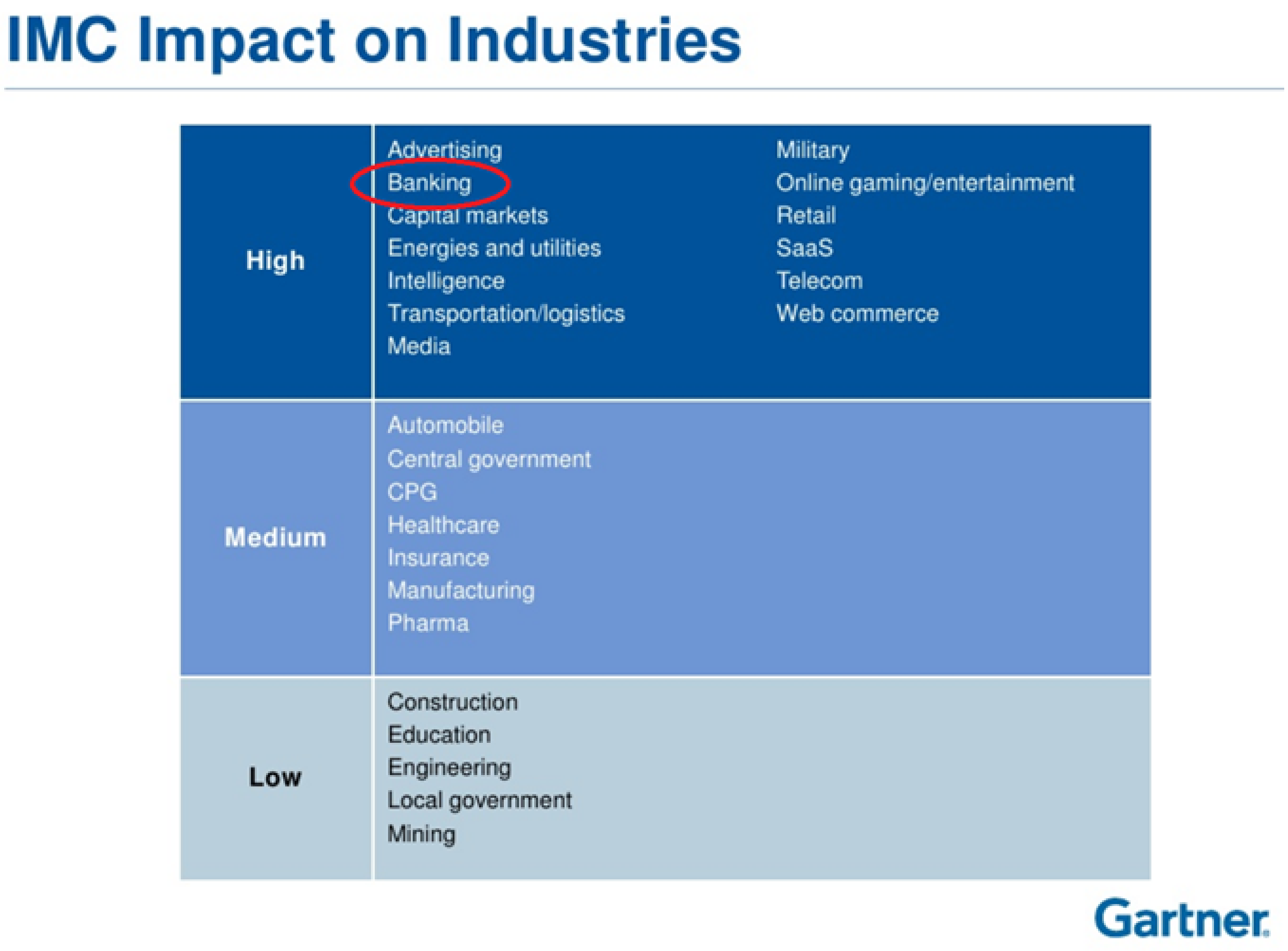
**Why Now?**

- 64-bit processors can address **up to 16 exabytes of data**
- DRAM production costs **drop by 32% every 12 months**
- 1GB of NAND flash memory **average price is 56$ cents***
- Commodity hardware provide **multi terabyte of DRAM**
- In-memory-enabling **software is available and proven**
- IMC software is often **embedded in products/services**

* Per Gartner's "Weekly Memory Pricing Index, 21 December 2012," G00247628

Gartner.

# Why an in-memory store?

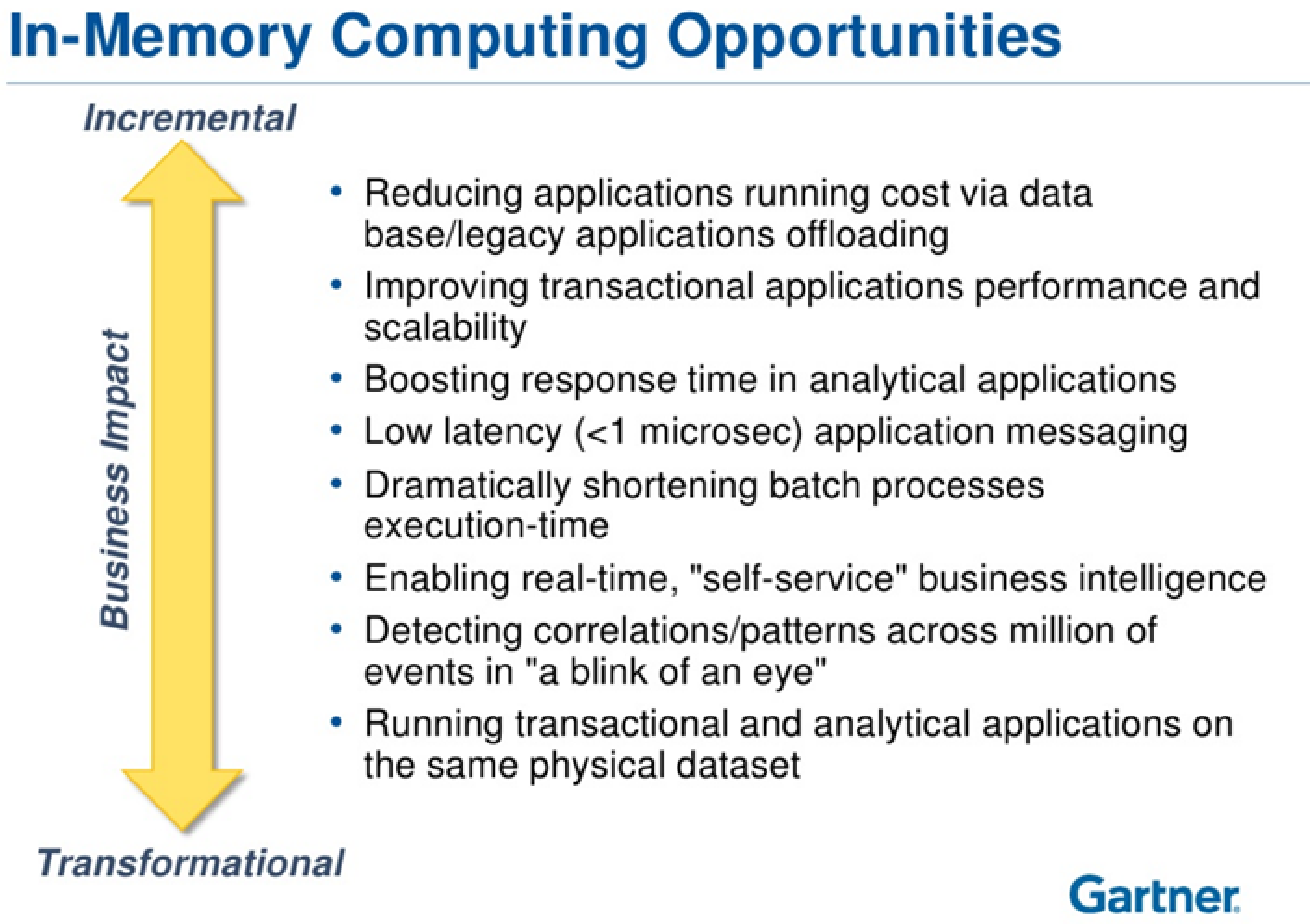


## IMC Impact on Industries

| | | |
|---|---|---|
| High | Advertising<br>Banking<br>Capital markets<br>Energies and utilities<br>Intelligence<br>Transportation/logistics<br>Media | Military<br>Online gaming/entertainment<br>Retail<br>SaaS<br>Telecom<br>Web commerce |
| Medium | Automobile<br>Central government<br>CPG<br>Healthcare<br>Insurance<br>Manufacturing<br>Pharma | |
| Low | Construction<br>Education<br>Engineering<br>Local government<br>Mining | |

Gartner

# Why an in-memory store?



## In-Memory Computing Opportunities

*Incremental*

*Business Impact*

*Transformational*

- Reducing applications running cost via data base/legacy applications offloading
- Improving transactional applications performance and scalability
- Boosting response time in analytical applications
- Low latency (<1 microsec) application messaging
- Dramatically shortening batch processes execution-time
- Enabling real-time, "self-service" business intelligence
- Detecting correlations/patterns across million of events in "a blink of an eye"
- Running transactional and analytical applications on the same physical dataset

Gartner.

# Technical advantages of an in-memory store

- Latency numbers every programmer should know:

L1 cache reference ......................... 0.5 ns

Branch mispredict ............................ 5 ns

L2 cache reference ........................... 7 ns

Mutex lock/unlock ........................... 25 ns

Main memory reference ...................... 100 ns

SSD random read ........................ 150,000 ns = 150 µs

Round trip within same datacenter ...... 500,000 ns = 0.5 ms

Disk seek ........................... 10,000,000 ns = 10 ms

Send packet CA->Netherlands->CA .... 150,000,000 ns = 150 ms

- Bandwidth numbers:

Read 1 MB seq. from memory ................ 20,000 ns = 20 µs

Read 1 MB seq. from FiberChannel ......... 400,000 ns = 400 µs

Read 1 MB seq. from SSD ................ 2,000,000 ns = 2 ms

Read 1 MB seq. over 1 Gbps network .... 10,000,000 ns = 10 ms

# Technical advantages of an in-memory store (Continued)

Read 1 MB seq. from disk ............. 20,000,000 ns = 20 ms

- Disks suffer from physical seek contention.
- Networks suffer from line/router contention.

# Motivation

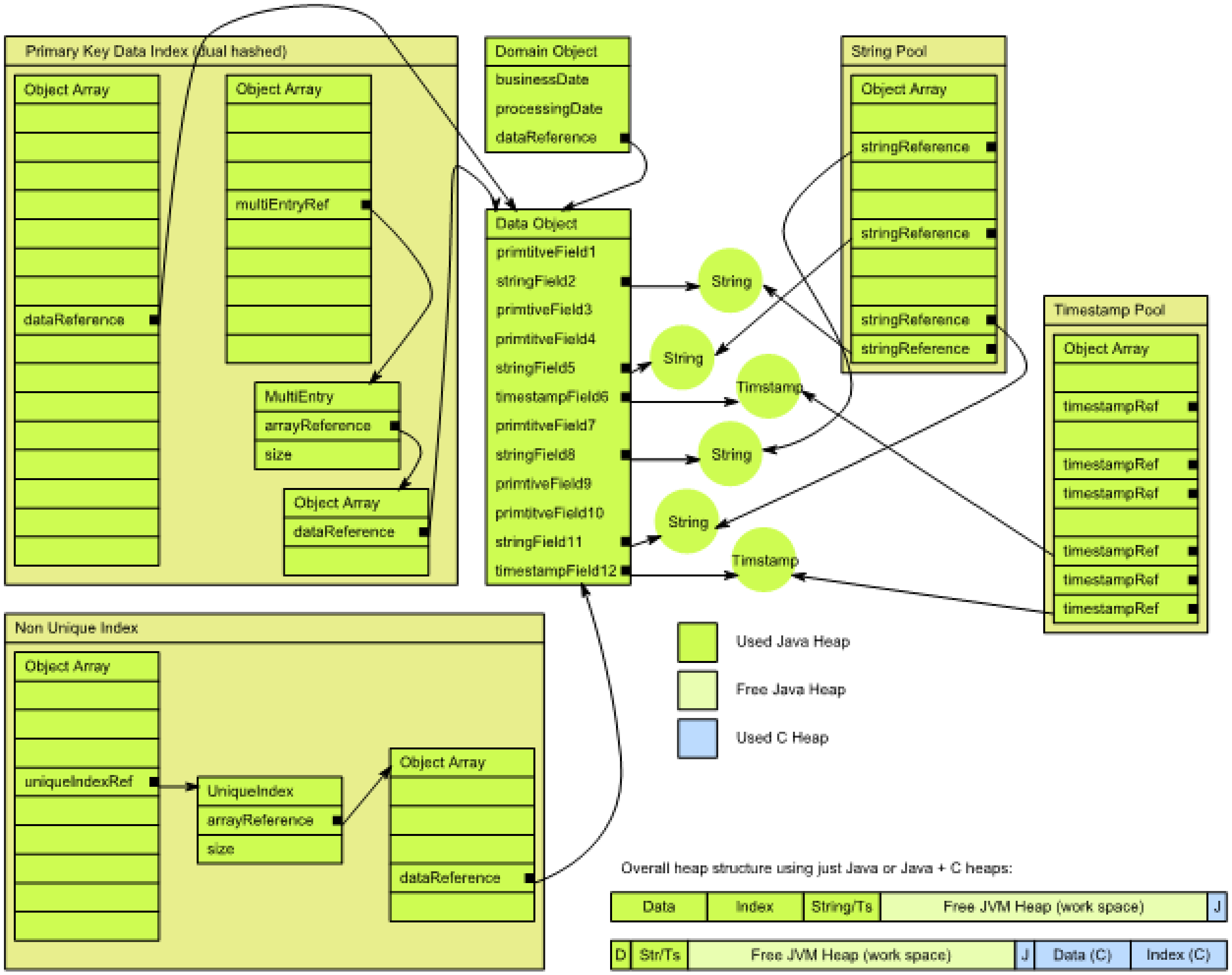
- A low collapse factor aggregation where the aggregation can happen along any dimension is such an example.
- GC – a Full GC can bring all processing to a halt
- Low resiliency: recover time can be very large.
- A replicated cache is more resilient: if a replica goes down, other replica can still function.
- Replication adds horizontal scalability to a system. Queries should be load balanced across replicas.
- Recovery is much faster: it’s faster to shutdown/restart a replica than reload from a database or cache archive.
- Since only the master cache interacts with the database, the database load is lower than having all caches talk to the database.
- Certain problems are best solved in memory.
- Storing large amounts of data in memory causes some issues.
- Replication has the following benefits:

# When is replication a good fit?

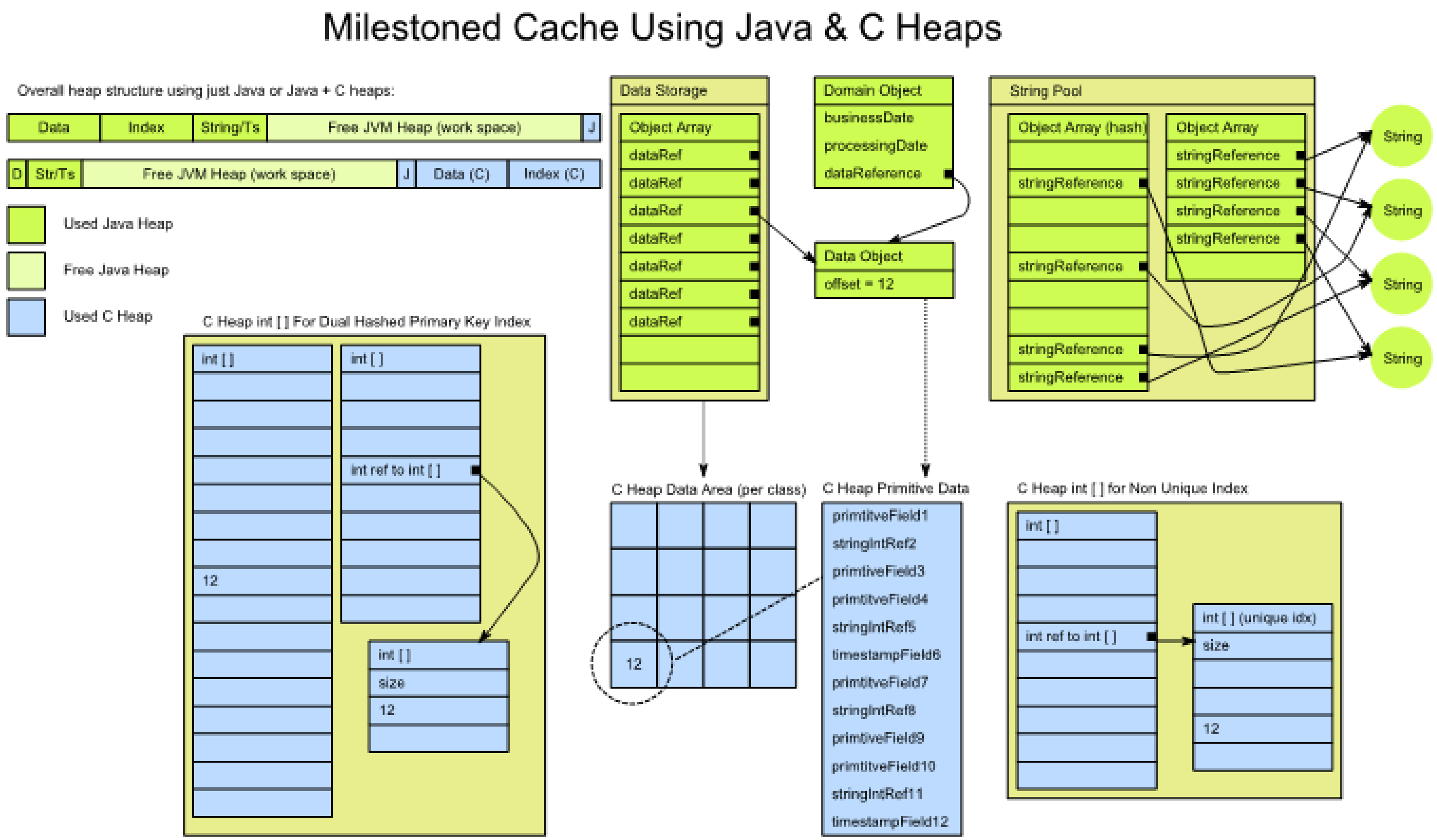
- Heavy read, no write (to the replicated classes).
- Can load balance queries between replicas.
- Large amounts of data (many GB).
- Configured for C-heap (only dated objects can go on the C-heap).
- Data is loaded via the Mithra CacheLoader.xml. Master cache is configured for periodic refresh from the database.
- All queries have a non-infinity processing date (aka snapshot). 9
- An application using replication should fit the following profile:

# Reladomo Java-heap only memory layout

Milestoned Cache Using Pure Java Heap



# Reladomo C and Java-heap memory layout



Milestoned Cache Using Java & C Heaps

# C-heap data storage and Strings

- E.g. the first 4 bytes is an integer account id; the next 8 bytes is the double quantity, etc.
- This can easily happen if the replica has non-replicated classes or multiple masters.
- MithraData objects are stored on the C-heap in a contiguous chunk of memory (per class).
- Each data object occupies a fixed number of bytes in a fixed format.
- Strings are not primitives and can’t be stored on the C-heap.
- Reladomo assigns a unique integer id to every String and stores this on the C- heap.
- A simple String array is kept on-heap. The array index of a String is the same number that’s stored on the c-heap for all occurrences of that String.
- What makes Strings and replication tricky is that the integers assigned to the Strings in the replica may not be the same as the ones on the master!

# Replication data organization

- MithraData objects are stored on the C-heap in a contiguous chunk of memory (per class).
- We only replicate the object data and not the indices.
- Every 1024 objects are called a*page*.
- Every page has a long page version. Version "0" is reserved for "dirty".
- On the master, any write to the C-heap marks the corresponding page as dirty. Replicas are read-only; the only writes are through replication.
- Each class has a long*currentPageVersion*.
- Every time a call from a replica arrives, the master cache is locked, the currentPageVersion is incremented, all dirty pages are assigned the new currentPageVersion, and finally the cache is unlocked.
- One of the core implementation criteria for replication is that there must never be any cache locks while data is sent on the network.
- Each replica knows the last synced currentPageVersion (per class). This value is included in the call to sync.

# Replication data organization (Continued)

- Conceptually, the master looks for pages that have a higher page version and sends only those.

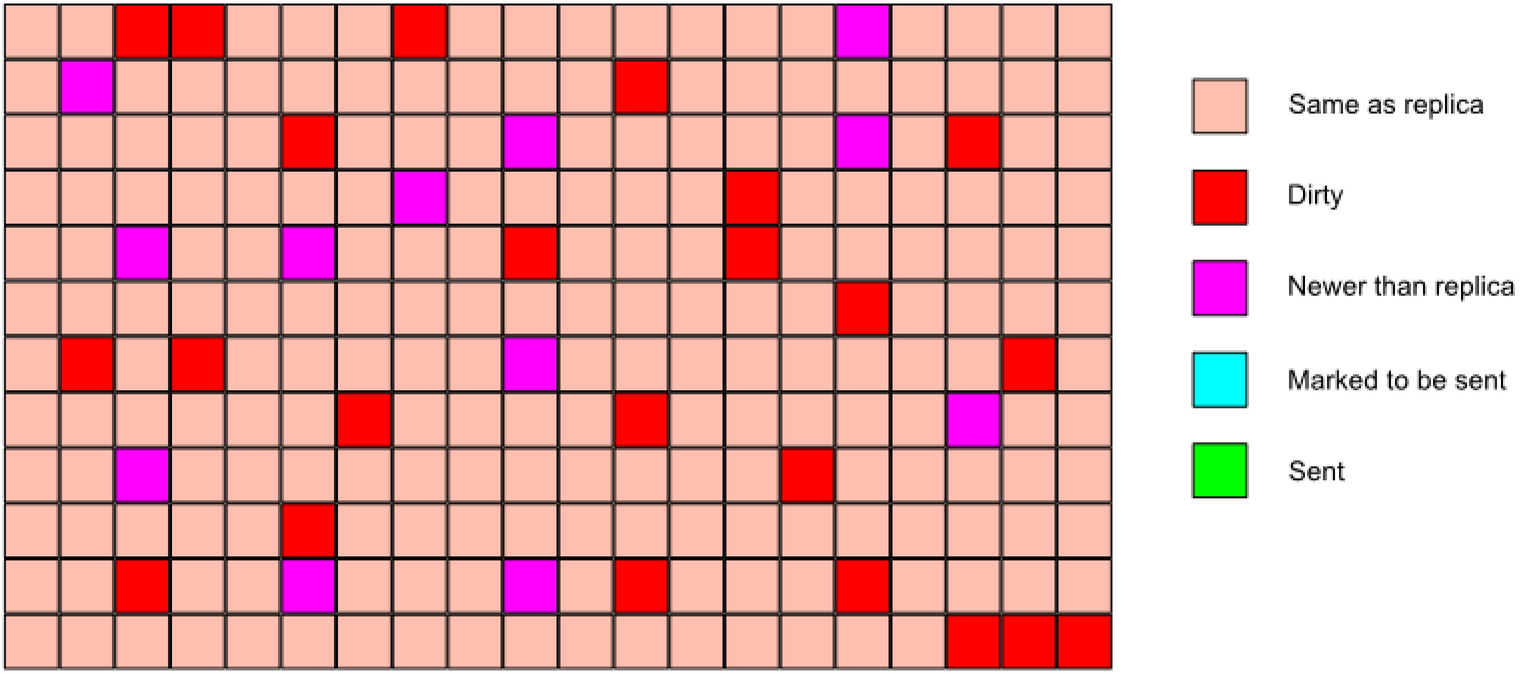
# Replication data stream

- The data stream between the master and replica caches is organized in terms of pages.
- The replica calls into the master cache, telling it what the max page version is that the replica has synced to.
- The master cache looks for any pages that have a higher page version and sends them to the replica.
- The tricky part is to do this without holding a lock while transferring data and still keep the data consistent.
- The main solution is to hold a lock, copy data elsewhere, release the lock and then transfer the copied version.
- To avoid using large amounts of memory, the amount of data copied is limited to 10 pages.
- There is a separate call to sync strings.
- MithraCacheUplink holds a map of the local (replica) integers to the master integers.
- When data arrives at the replica, the string integers are replaced with the local value.

# Master sync algorithm

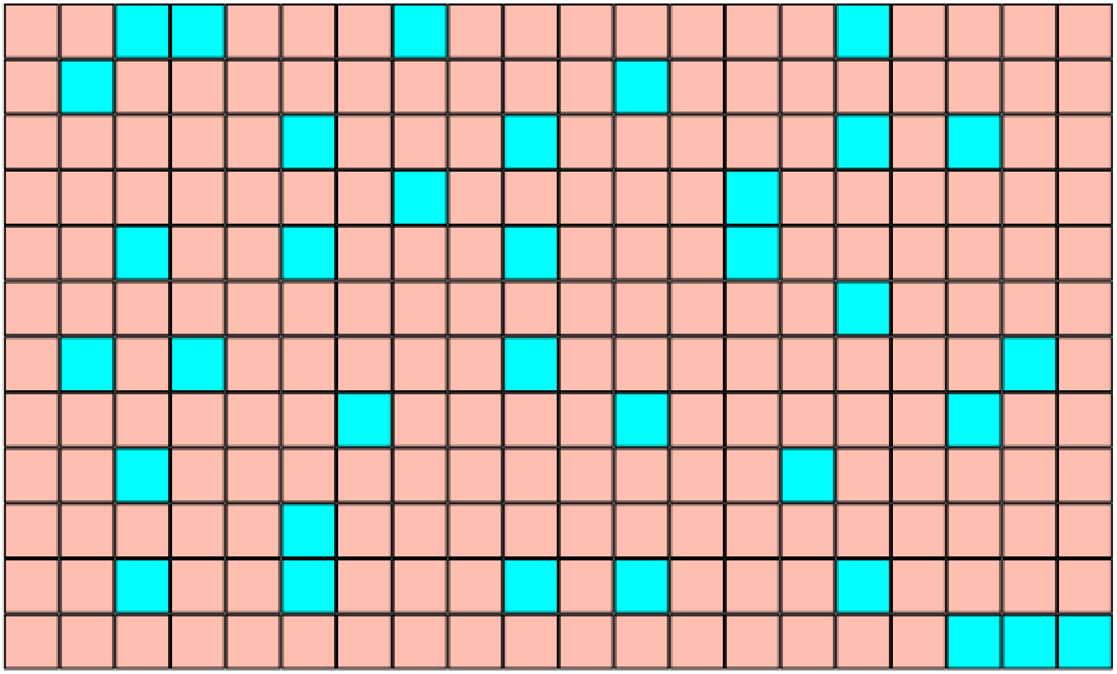
- Under a lock, the pages that have to be sent are computed.
- If the number of pages to send is less than 10, the pages are copied (still under lock) and the lock is released. The copied pages are sent and we're done.
- If the number of pages to send is greater than 10, they are sent in batches.
- For each batch (10 pages), the cache is locked, we look at the page versions for that batch, if nothing has changed, we copy the pages, release the lock and send the pages.
- If something has changed, we re-scan the cache for pages that need to be sent and restart from the beginning.
- A replica can therefore see the same page more than once in its incoming stream. It'll ignore the older copies.
- See the algorithm in FastUnsafeOffHeapDataStorage.serializeSyncResult

# Heap State before Sync



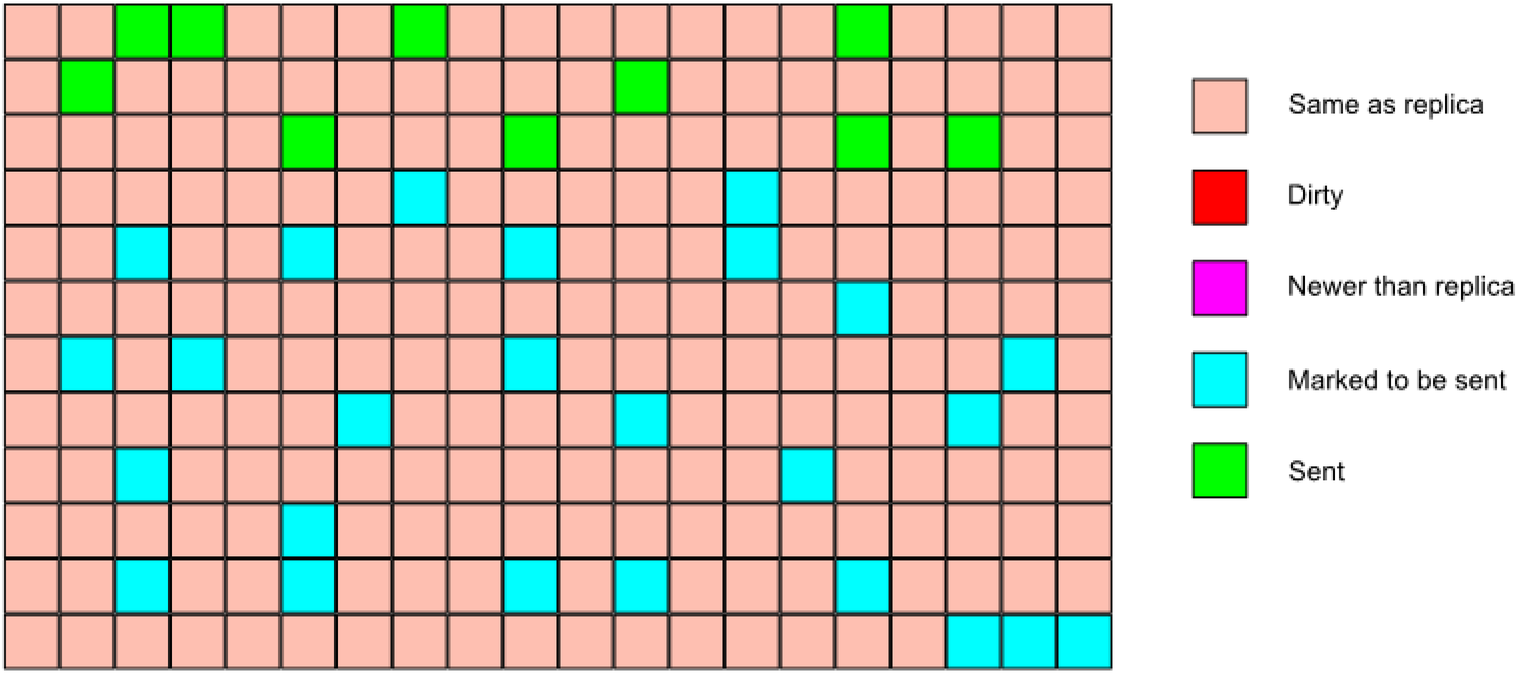
- 🟧 Same as replica
- 🟥 Dirty
- 🟪 Newer than replica
- 🟦 Marked to be sent
- 🟩 Sent

# Initial Page Scan

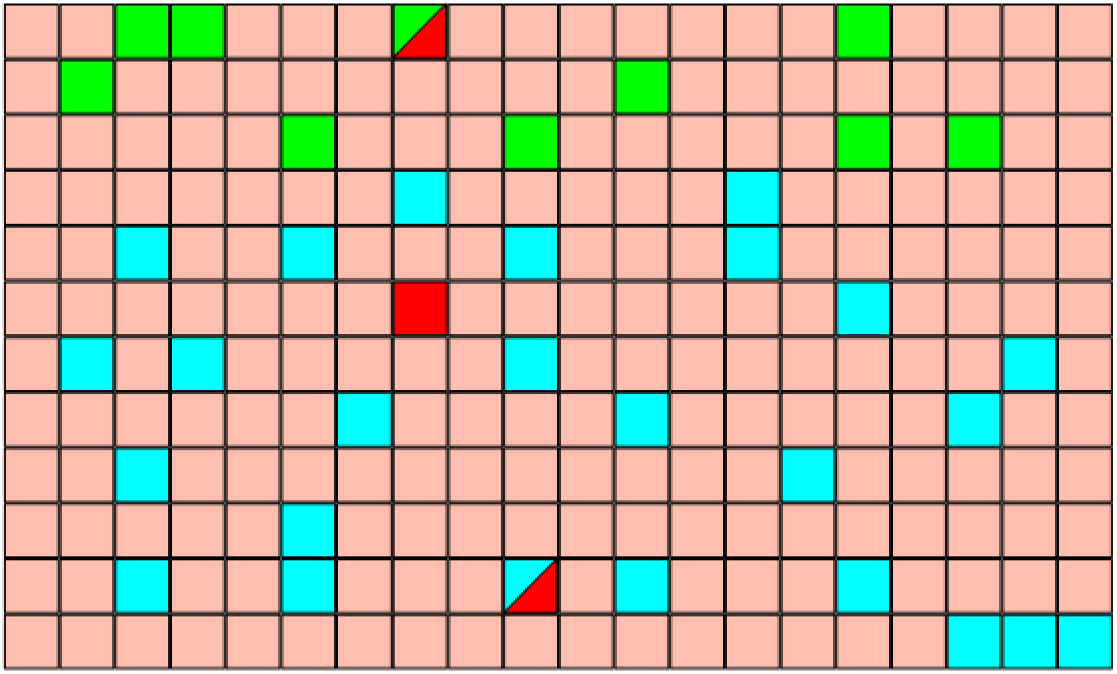


- 🟧 Same as replica
- 🟥 Dirty
- 🟪 Newer than replica
- 🟦 Marked to be sent
- 🟩 Sent

# First Group Sent



# Cache is updated during first group send



🟫 Same as replica

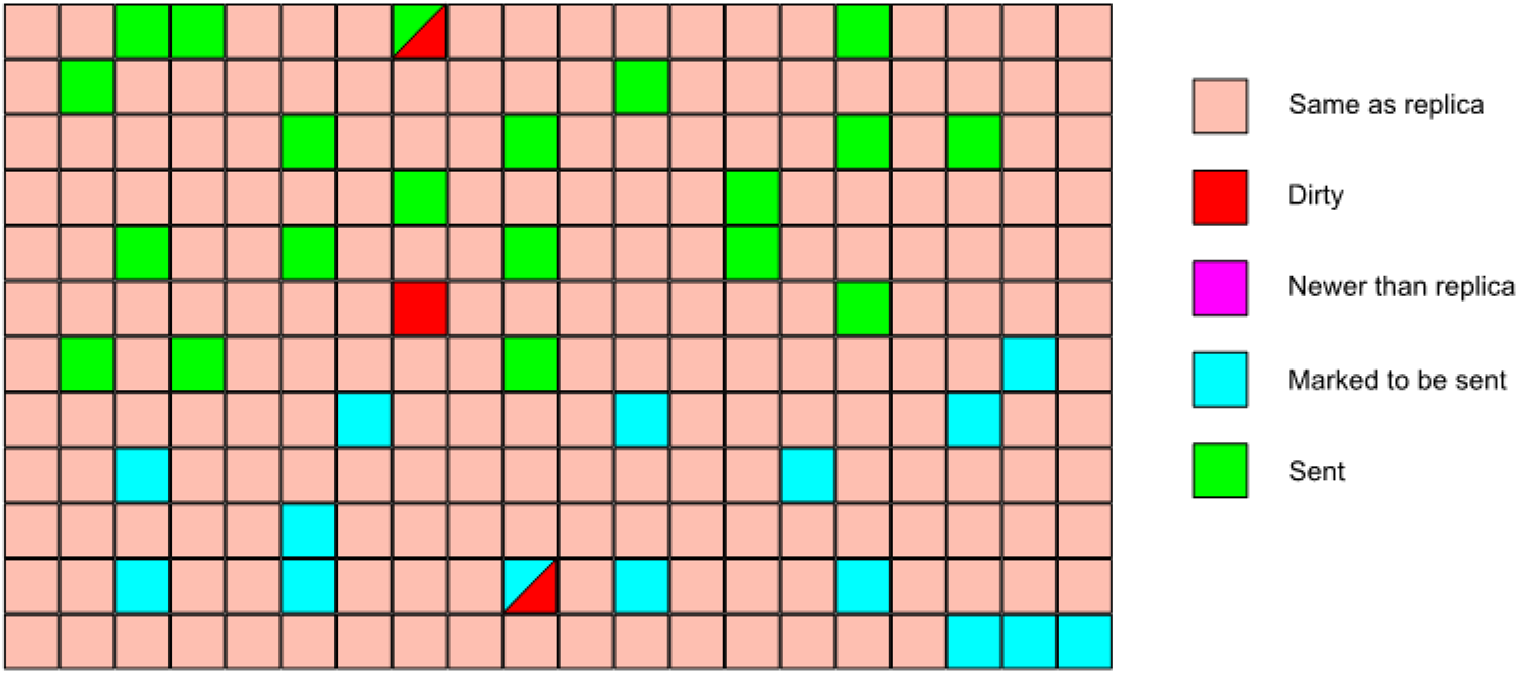
🟥 Dirty

🟪 Newer than replica

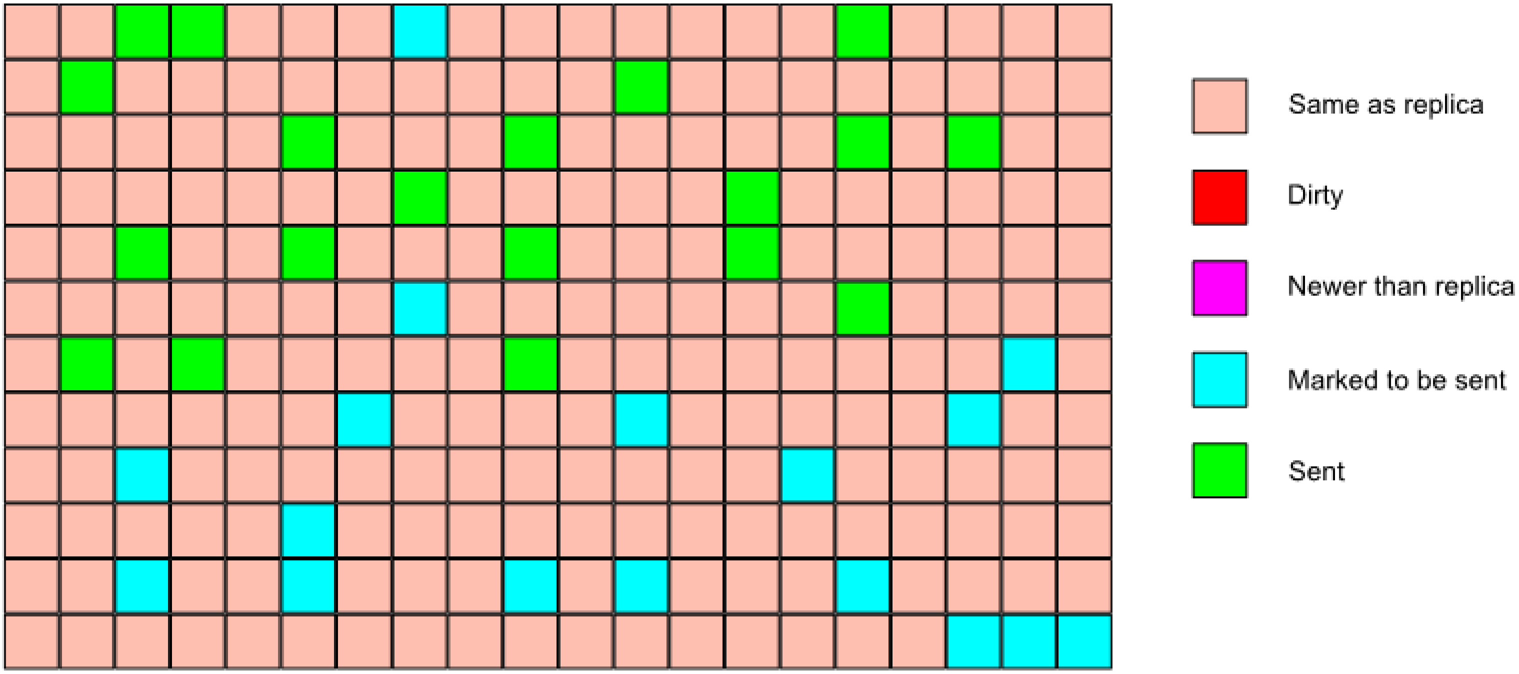
🟦 Marked to be sent

🟩 Sent

# Second Group Sent

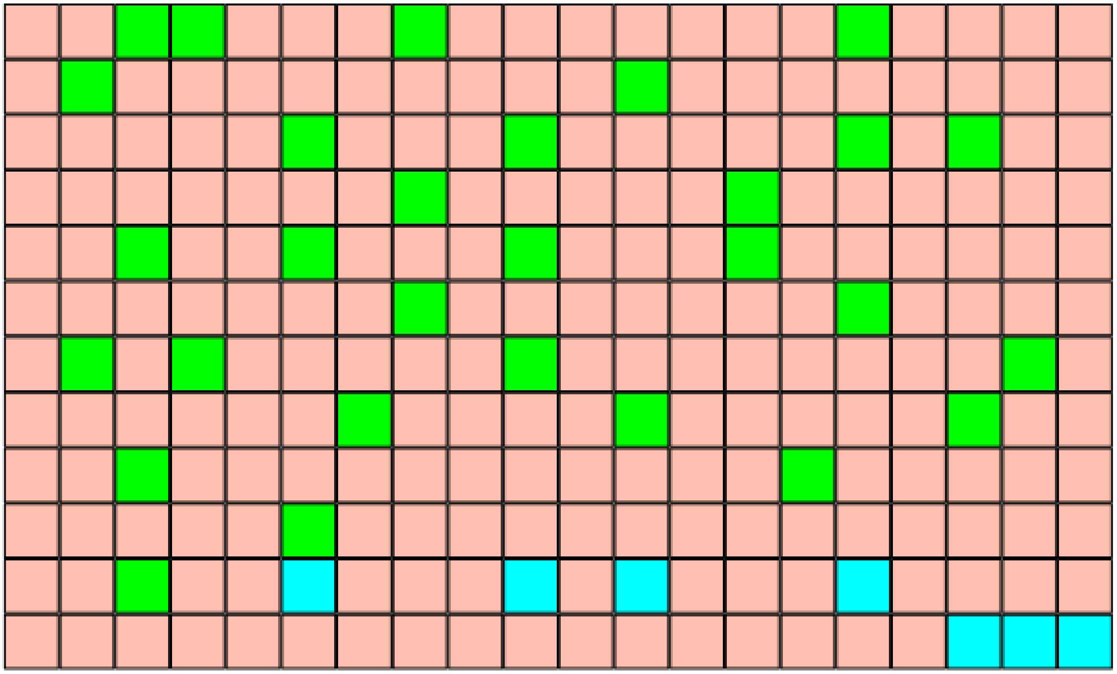


# Third Group Rescan



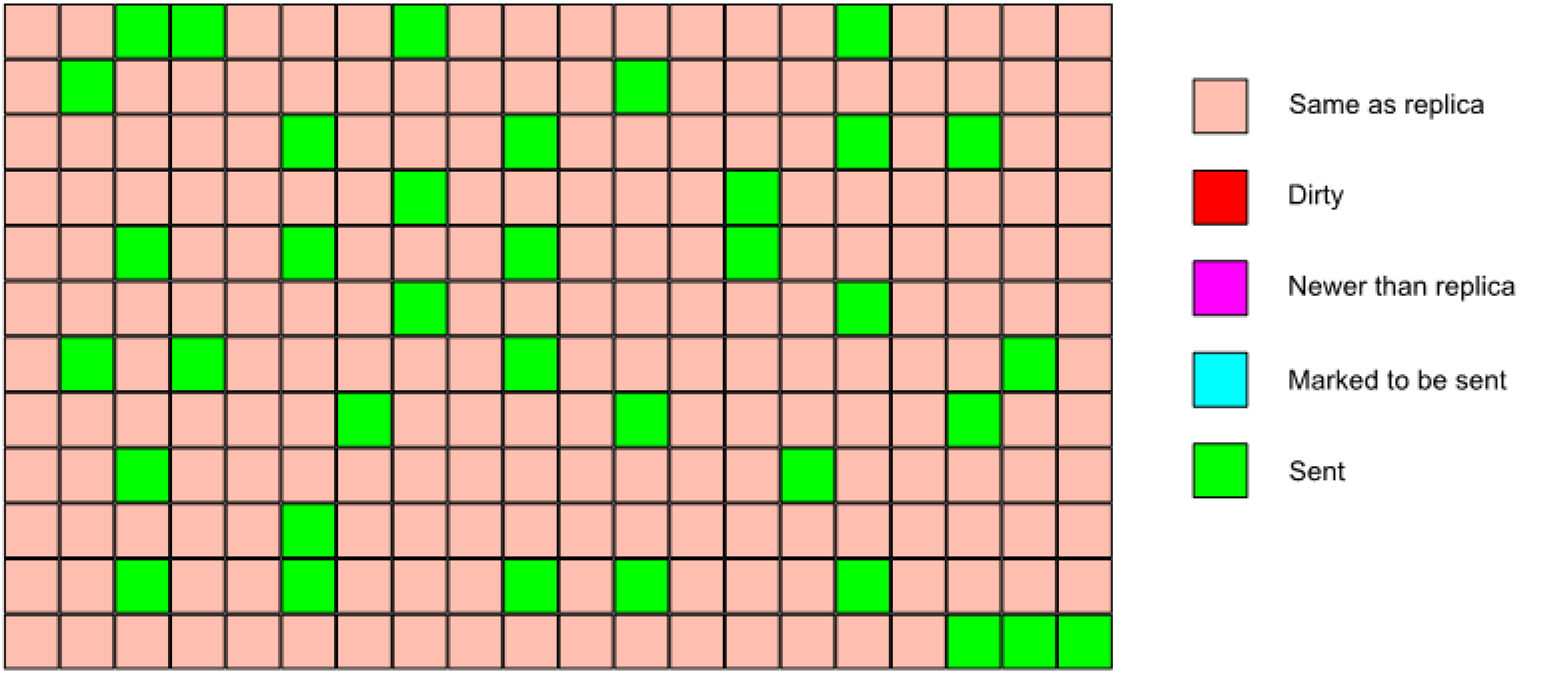
- Same as replica
- 🟥 Dirty
- 🟪 Newer than replica
- 🟦 Marked to be sent
- 🟩 Sent

# Third Group Sent



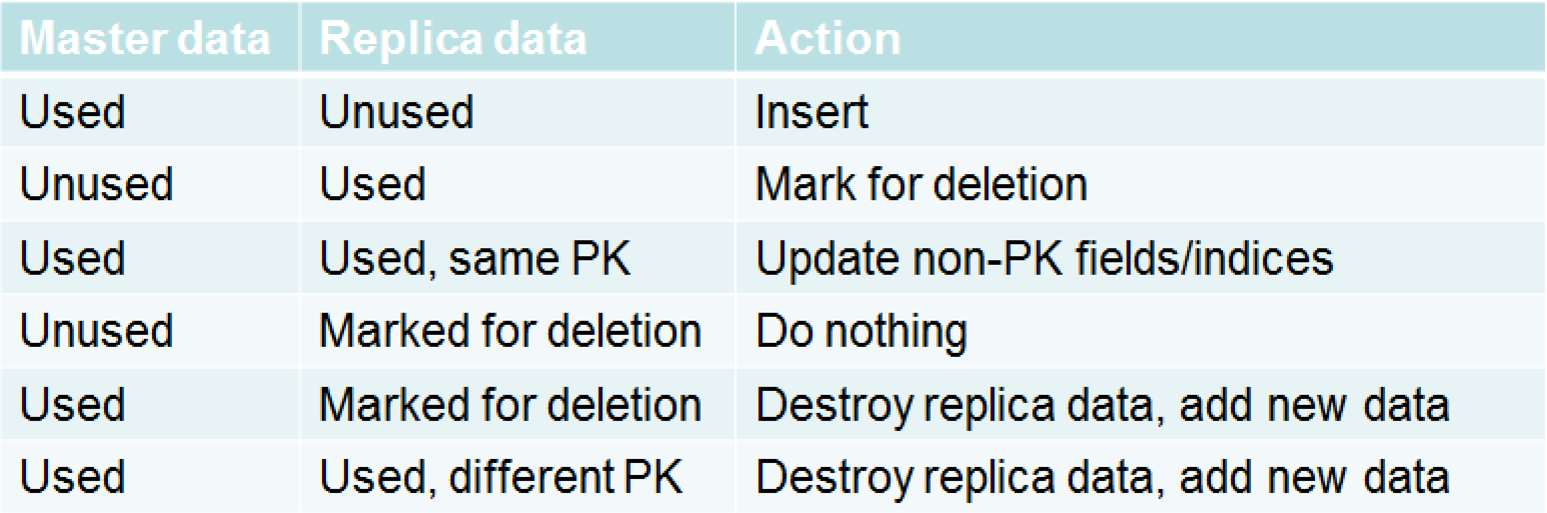
- 🟫 Same as replica
- 🟥 Dirty
- 🟪 Newer than replica
- 🟦 Marked to be sent
- 🟩 Sent

# Fourth Group Sent



# Replica sync algorithm

- The replica receives a set of pages from the master.
- Each page has a page number, page version, page data and used data array.
- The used data array is a bit set representing which data slots on a page are actually used.
- The replica then has to compare the incoming page with the exiting page (if any) and correctly update its local cache indices according to the modified data.



| Master data | Replica data | Action |
|---|---|---|
| Used | Unused | Insert |
| Unused | Used | Mark for deletion |
| Used | Used, same PK | Update non-PK fields/indices |
| Unused | Marked for deletion | Do nothing |
| Used | Marked for deletion | Destroy replica data, add new data |
| Used | Used, different PK | Destroy replica data, add new data |

# Removing data from the master cache

- The application will disallow any further queries involving the data.
- It will then wait for all running queries to finish.
- Finally, it can remove the data from the master.
- It is occasionally useful to remove data from a cache (e.g. drop a date).
- It is critical that the data not be currently in use on any replica.
- This has to be implemented at the application level. The typical implementation uses a quiesce algorithm:
- Removed data is marked for deletion on the replica. The data object is only
- Once the GC removes the reference, the data is released and the slot is marked as unused.
- If the GC cycle happens faster on the master, it’s possible that the data on the replica is not fully released yet.
- If new data is added to the master, its possible that the data on the replica has to be thrown away (destroyed).

# Removing data from the master cache (Continued)

- Any destroyed data will throw exceptions if accessed. That’s why it’s critical the application has safeguards against referencing deleted data.

# Restarting Master

- A new master is started.
- A single replica is quiesced (no new queries are allowed, existing queries run to completion).
- The replica is then shutdown and restarted pointing to the new master.
- The steps are repeated for other replicas until they are all on the new master.
- Queries can be served from either the new or old replicas while the procedure is continuing.
- The data synchronization between the master and replica is based on the physical layout of the data on the master.
- The physical layout of the data can’t be easily reproduced if the master goes down.
- Therefore, when a master goes down, the running replicas cannot switch to a new master. A replica that has lost its master is called a ronin.
- To restart the replica, the following procedure must be implemented:

# Milestone Consistency

- How do we query the replicas (that may be in different sync states) and get consistent results?
- It's just as important that the trade data and product data it joins to are consistent
- How can this be done with a query against a database?
- How can this be done with a query against a singular cache?
- How can this be done with replication?
- Central question:
- The question exists even when we don't have replication, because we read many different types of data out of the database
- We'll cover this in three steps:

# Milestone Consistency: Queries and Mutation

- The query must refer to a fixed point in time that is guaranteed to never mutate.
- "now" is not fixed, but "now - 2 minutes" is. We usually add a little bit for clock synchronization, e.g. "now- 2.5 minutes", which we'll call *"stability time"*
- In a system that implements audit milestoning, we have to understand when a query is guaranteed to be repeatable.
- All mutations occur within the context of a transaction.
- All objects in the same transaction have the same mutation time, which is taken to be the transaction start time.
- "now" is not a fixed point (yet), because there may be transactions in flight that have started and not finished yet.
- It is imperative for all transactions to have a guaranteed timeout. 2 minutes is our production setting.
- So when doing queries against the DB, a query of the form IN <= processingTime && > processingTime is guaranteed to be correct if processingtime less than stability time.

# Milestone Consistency: Master Cache

- Wall clock: 4:11:00 pm
- DB stability time: 4:08:30 pm
- Cache stability time 4:00:00 pm
- at 4:11, cache asks for data between 4:00:00 and 4:08:30. refresh finishes at 4:15. cache's stability time is moved to 4:08:30
- Example:
- A cache (master or stand alone) maintains its own stability time (sometime called "now snapshot").
- When a cache performs a refresh, it asks for all data newer than its current stability time, but older than database stability time. when the refresh is finished, the cache's stability time is moved forward.

# Master & replica database refresh time

- It’s possible different classes have different refresh times on the replica because the sync guarantee is on a per-class basis.
- A replica may also have multiple masters, which will naturally have different refresh times.
- Reladomo cache loader has a periodic refresh.
- This refresh is running on the master.
- A refresh cycle picks a refresh time, does a database refresh across all classes and marks them with the new refresh time.
- As part of the master-replica sync, this refresh value is sent to the replica (per class).
- The replica can then compute the minimum refresh time from all its classes to arrive at the replica refresh time.
- The application must be aware of this refresh time and construct all queries accordingly (processing date must not be greater than the refresh time).

# Performance numbers

- 100 GB cache, takes 3.5 hours to load from database.
- 10.8 GB compressed cache archive.
- About 24 minutes to read the cache archive on a bare metal.
- BM master to BM replica takes about 8 minutes to sync with 15 threads.
- Subsequent refresh syncs take a few seconds.

# Conclusion

- Single point of failure.
- Time to recover.
- Servicing a large number of requests from many users/systems.
- Dealing with application growth over time.
- Resiliency
- Processing power
- C-heap storage
- Proper milestoning
- Smart loadbalancing
- Proper refresh
- It has to be combined with
- As we solve the problems of large Java memory foot print, we start to run a new set of problems.
- Replication can be an effective way to address these issues.

# Appendix: Configuring Replication

- That usually means adding a PSP service that uses MasterCacheService/Impl for its interface and implementation.
- A master cache is configured by enabling MasterCacheService on the server.
- A replica cache is configured by adding MasterCacheReplicationServer to the runtime configuration.

-
  ```
  <MithraRuntime>

    <MasterCacheReplicationServer masterCacheId="mithra.tes

    className="com.gs.fw.common.mithra.test.util.PspBasedMi

    syncIntervalInMilliseconds="2000"
  />
  </MithraRuntime>
  ```
- The replica will create a MasterCacheUplink for every configured master.

# Appendix: Configuring Replication (Continued)

- The job of the MasterCacheUplink is to use the remote MasterCacheService and keep the replica in sync with the master.
- Periodically, the uplink will poll the master cache to get new updates.
- The process of copying the data to the replica is called "sync".
- During each sync, the uplink uses a fixed (configurable) number of threads to poll the master.

# Appendix: MasterCacheService

- Used for getting the initial configuration from the master cache.
- The main method that gets called periodically for each class.
- Strings need special handling. More on this later.
- Used to optimize the initial sync. Larger classes are synced first.
- public RemoteMithraObjectConfig[] getObjectConfigurations();
- public MasterSyncResult syncWithMasterCache(String businessClassName, long maxReplicatedPageVersion);
- public MasterRetrieveStringResult retrieveStrings(int startIndex);
- public MasterRetrieveInitialSyncSizeResult retrieveInitialSyncSize();