Database

LECTURE -11

Strong Consistency in cache Augmented SQL Systems.

→ Simple Operations.

Predominatly, caches are used to support simple operations; read & write a small amount of data, Require low latency; meaning that a human being is issuing these operations & cannot wait too long,

→ Slow RDBMS.

The resources are being utilized to implement the core functionality of the DBMS & it's slowing down these applications.

Solution: Augmention with:
(to hide the latency)

— a lean & mean look aside cache that is using memory only.
— LRU / CAMP eviction
— Network protocol for key-value transmission.

→ Cache Augument DBMS

When a request comes in, it goes to the cache & asks for the user profile; If the cache returns a value, then uses the value to generate HTML result page.

If the cache reports a miss, meaning that it doesn't have this key value pair, then the application, is forced to issue SQL queries to the relational database management system process the result of those queries & construct a value using the result of the queries.

The more writes we have, lower the score of the cache augmented architecture.

→ Terminology.

A Session is a sequence of RDBMS operations as 1 transaction followed by one or more key-value store (KVS) operations.

— The order is not important, whether you do transaction first followed by KVS operations.

— A session acquires leases in a manner similar to 2-phase locking.

A key-value stores, refers to the cache.

3 approaches to update the KVS (cache)

☆

→ Incremental update.

Put a counter in cache & you issue plus one to it, essentially a stream of plus ones are coming into the cache & that counter keeps incrementing, & then the application looks up the value & presents it to whoever needs it.

→ Invalidate.

It would essentially delete that counter everytime there is a change.

→ Refresh.

It would read the value incremented by one & then write it to the KVS.

If there's only one session running in the system, with BG workload, there is no unpredictable data, But as soon as we have concurrent sessions being issued to the system, suddenly there is unpredictable data, & the amount increases as a function of the number of concurrent sessions.

↑ because the architecture incurs undesirable race conditions that cause it to produce this unpredectable or stale data.

★ → Invalidate.

- The key for the cache entry; which is profile & inviteeID & it deletes that key-value pair from the cache.
- It the begins the RDBMS transaction, inserts the inviteeID in the Pending Friends, & updates the pending friend count by one in the user table & then commits transaction.

* By deleting the cache entry, it's essentially forcing the next request that references that cache entry to go to the database management system & compute the missing entry. And by doing so it maintains the cache & DBMS consistent with one another. Deleting the same entry is not considered as a conflict.

→ Refresh / Refill

Here, instead of deleting the cache entry; it constructs the key.
Reads the old value → increments the pending friends by one → it writes back into cash

→ Incremental update.

Here, the instructions are replaced by an operation that's supported by the cache, the KVS; called Increment.

So, we just construct the key issue the KVS command increment, key & then the delta, it knows how to increment the value of the key by one. ↑1

* Similar to the SQL update command.

→ Strong Consistency.
The consequences of violating strong consistency is that the number of friends don't match the list.
eg: reverse ordering of comment & its responses.
A shopping cart with extra items.

Key insights: The RDBMS is not aware of the cache & the application inserts stale data in cache due to → Undesirable race conditions.
→ Snapshot isolation.

→ Undesirable race conditions.
May result in cache states that are inconsistent with the database state.
If the operations are interleaved

→ Snap shot isolation.
MVCC may insert stale data.

This has nothing to do with the cache theorem, there's no network partition, no failed server, it's really the case that the software / architecture is causing this stale data to be generated in the cache.

Preventing these limitations simplifies software development by making the behavior of programs more predictable. (one implementation for all application use cases.).

Instead of a programmer sitting & trying to think about all these possible race conditions, it would be ideal to have a framework that can take care of these undesirable race conditions.

→ Approaches to address this limitation.
Modify the key-value store (KVS) only.
Modify the RDBMS only
Modify the RDBMS & KVS
One of the first 3 in combination with the application.

IQ framework focuses of this because the software is very simple compared to the software for RDBMS

→ Modify KVS only
Changes to
- KVS Server
- KVS client

Advantages:
Functions with diverse RDBMS including proprietary ones; eg: Oracle, SQL, etc.
KVS components are simpler software systems than RDBMS.
Enables transparent caching; eg: KOSAR.

→ Session.

It's trying to implement the concept of a transaction across both the key value store & the relational database management system.

Logically, this session imposes the serial schedule observed by the relational database management system onto the cache manager.

A session is just a sequence of all RDBMS operations as one transaction, & one or more key value store operations.

A sessions update should be visible to itself until it commits.

→ IQ Framework with Invalidate

Inhibit lease ↙

If any existing I lease on a key-value pair, if a requester comes along & asks for I lease, the requester is asked to back-off & try again.

used by KVS look up for a cache entry & granted when there is a miss for the referenced key ↘

The Quarantine lease is used by a writer, asks for a Q lease for write purpose, the server grants the lease & avoids I lease.

If the requester comes in asking for an I lease, it's not compatible & so the requester is asked to back-off & try again.

If there is a quarantine lease & the requester wants to write it, it's granted & it's allowed proceed forward.

LOCK	LEASE
→ It has an infinite lifetime & an entity must unlock a key-value pair	→ Finite life time & lifetime expires, the lease is relinquished.
	→ When the lifetime is short it will result in an empty cache.
	→ If the lifetime is too long, it starts to lock like a lock & its bad because the data may start to became unavailable.

→ How is I-lease implemented?

A lot of design details may be implemented in the KVS client & are hidden from the application IQ-Servers represents a lease as a token generated by KVS server. IQ client maintains the lease granted by the KVS server.

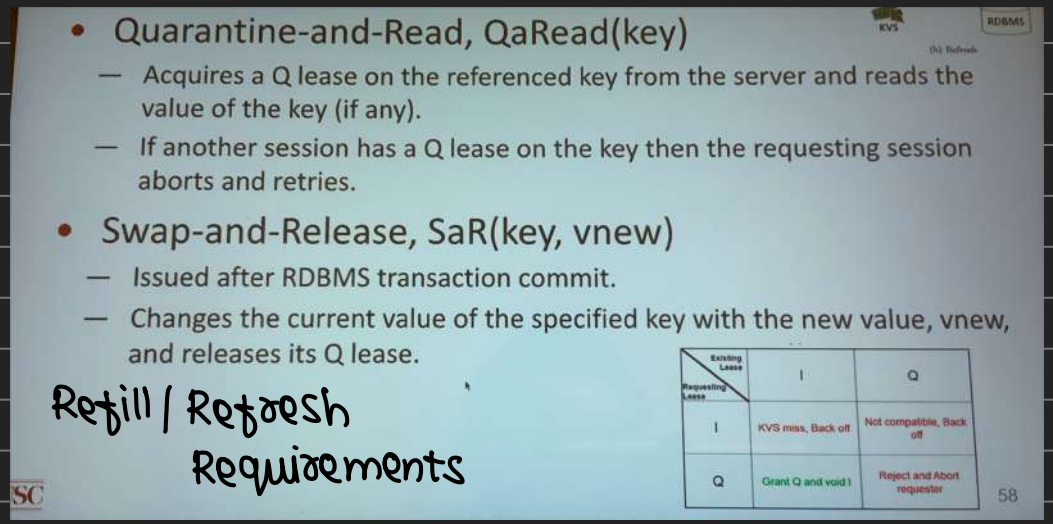
Argument issued by application → IQ client forwards the I lease to → IQ server as an argument of the set command

The Q-lease is released when the session commits or aborts.

↑ Applies to both I & Q lease.

Refresh & Incremental VS Invalidate.

We don't want a session to consume values, that are dirty / non-existent because of the session aborted.

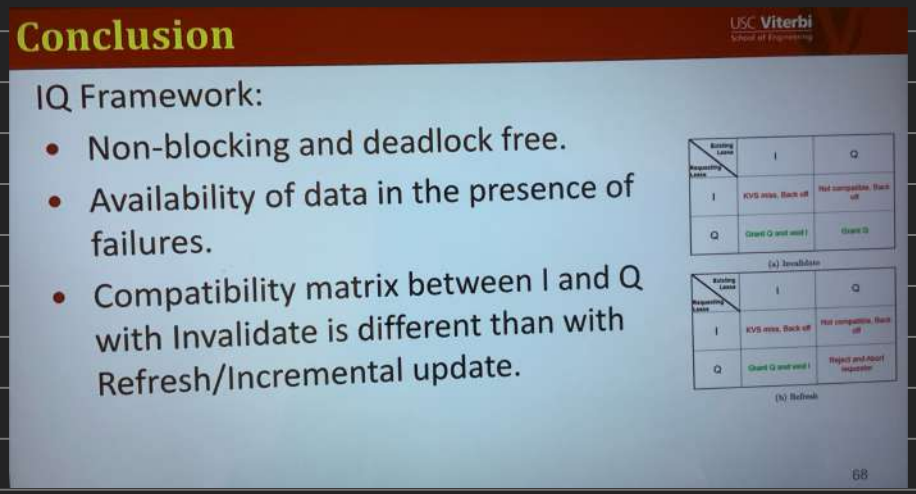


Write sessions that modify the state of the DBMS and the cache should not be allowed to do cache look up, Instead they must issue DBMS queries. OR have a fat client.

→ Incremental update Requirements

Essentially its buffering things & it waits for the comments before it applies it, it could also do this buffering at the client side, to have a fat client.

→ What if we have I lease but not Q lease? Its going to generate stale data.
Using both I & Q lease we can eleminate stale reads that are coming out of the cache.



☆ → Thundering Reads
A thundering herd happens when a specific key undergoes heavy read & write activity.

I lease ← solution.
The first read for the specific key is granted the I lease, all the other reads observe a miss & back-off.
The read with I lease queries the RDBMS, computes the missing value & populates the cache & then the thundering herd observes the cache hit & the request are served expeditiously & efficiently.