SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

Database Systems

WS 08/09

Prof. Dr. Jens Dittrich

Chair of Information Systems Group
http://infosys.cs.uni-saarland.de

Topics (5/6)

- large systems
  - global scale data management
  - map/reduce
  - pig
  - search engines
  - data warehouses and OLAP
- write-optimized system concepts
  - OLTP
  - publish/subscribe
  - streaming
  - moving objects
- management of geographical data
  - basic concepts
  - GIS, google maps

Large Systems.

Introduction.

Large Systems

- Yahoo!
- Google
- Amazon
- All in the Petabyte range
- How do they store their data?
- Use some distributed DBMS product and scale it, right?

Large Systems

- Yahoo!
  - PNUTS
  - Sherpa Data Services
  - Pig
- Google
  - Google File System (GFS)
  - BigTable
  - map/reduce
- Amazon
  - Dynamo
- Do you see the word “DBMS“ here?

Why not use a DBMS?

- scalability in the Petabyte range?
- on ten thousands of nodes?
- transactions? for what?
- SQL?
- cost for licenses?
- flexibility?
  - optimized storage layout: columns, column groups, etc.
  - optimized write behavior: log strutured trees/stepped merge
  - design global-scale distribution and redundancy
  - can remove bottlenecks inefficiencies as they arise: full control

Common Strategy

- build system from scratch
- use existing data managing technology wherever possible
  - either re-implement:
    - does not exclude reinventing the wheel
    - give new names to existing things
  - and/or use DBMS or file system as a little component in a bigger architecture
- leave away unnecessary stuff
  - no SQL
  - no transactions
  - etc.
- adapt technology to work with a very large number of machines

Agenda

- Yahoo!
  - **PNUTS (today)**
  - Sherpa Data Services
  - **Pig (next week)**
- Google
  - Google File System (GFS)
  - BigTable
  - **map/reduce (next week)**
- Amazon
  - Dynamo

SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

PNUTS.

What is PNUTS?

- massively parallel and geographical distributed DBMS (sic!)
- used and built by Yahoo! for its Web applications
- first version out and being used
- proprietary system
- not just installing an existing DBMS product
- DBMSs are just small building blocks in this architecture
- makes use and recombines considerable DBMS technology
- Literature:
  Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, Ramana Yerneni: PNUTS: Yahoo!'s hosted data serving platform. PVLDB 1(2):1277-1288 (2008)

Requirements: Scalability and Response Time

- scalability
  - adapt to peaks and periods of rapid growth
  - be able to add new resources with minimal effort
- response time
  - applications must consistently meet Yahoo!‘s internal SLAs for page load time
  - =>response time requirements on data management platform
  - users scattered across the globe
  - => need replicas distributed around the globe
  - otherwise latency requirements cannot be met

Requirements: High Availability, Fault Tolerance

- high availability and fault tolerance
  - downtime means money lost
  - => cannot serve ads
  - => do not get paid
  - => disappoint users
  - service must survive
    - server failures
    - network partitions
    - power loss

Requirements: Relaxed Consistency Guarantees

- relaxed consistency guarantees
  - DBMS use model of serializable transactions (see week 10)
  - trade-off between performance and consistency
  - supporting serializability over a globally replicated and distributed system is very expensive
  - achieving serializability for this app impractical
  - also: transactions not really needed as most apps tend to manipulate one record at a time
  - for instance: user changes avatar, post new pictures, invites friends to connect
  - little harm is done if the new avatar is not initially visible to one friend, etc.
  - => apps may trade consistency for performance

Requirements: Eventual Consistency?

- eventual consistency?
  - model used by other distributed replicated systems (e.g. Amazon‘s Dynamo)
  - idea:
    - a client can update any replica of an object
    - all updates to an object will **eventually** be applied
    - but potentially in different orders at different replicas
- this model may be appropriate for some aplications
- however, this model is too weak for Yahoo!

Eventual Consistency Counter-Example

- consider a photo sharing application
- allows users to share photos and control access
- keep ACL for each user: determines who is allowed to see photos
- Update1: remove my mother from the list of people who can view my photos
- Update2: post spring-break photos
- Update1 can go to replica R1 while
- Update2 might go to replica R2
- mother may be able to see spring-break photos for a short time at R2 (as Update1 has not arrived yet)
- Upps!

Eventual Consistency Counter-Example

- model, however, guarantees that all updates are applied eventually
- still, example breaks contract with the user
- user does not want mother to see spring break pictures at **any** time
- root of the problem: order of U1 and U2 is not guaranteed!!
- => stronger guarantees are needed for these kind of apps
- transactions would do the job
- however, in this case there is another solution
- still it will be acceptable to read (slightly) stale data

PNUTS Overview

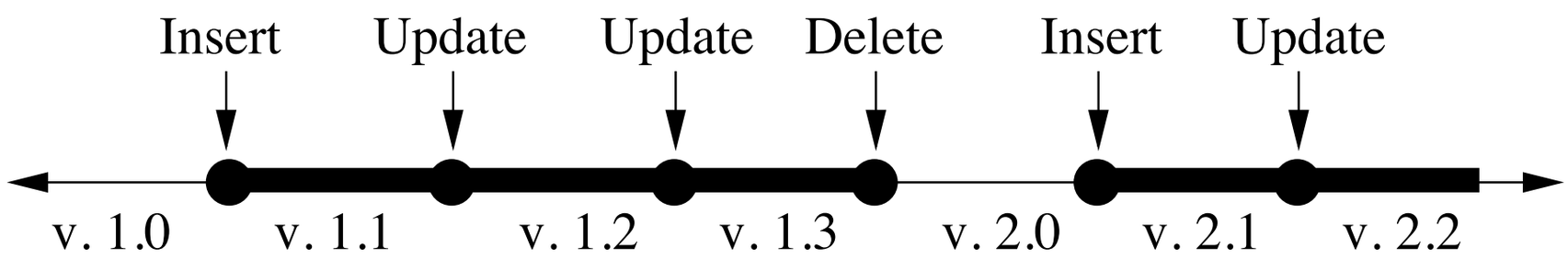
- data model and features
  - simple relational model
  - point and range queries
  - notifications
  - bulk loading
- fault tolerance
  - redundancy at multiple levels
- pub-sub message system
  - message broker as log replacement
- record-level mastering
  - all operations asynchronous
  - per-record timeline consistency
- hosting
  - centrally managed DBMS

Data and Query Model

- relational model
- tables of records with attributes
- blob is a valid data type (stuff in tables handled by apps)
- new attribute can be added at any time without halting queries or updates
- records are not required to have values for all attributes
- simple query language
- single table queries only
- no integrity constraints, joins, group-by
- tables may be hashed or ordered (point or range queries)
- expect point queries or small scans: few tens or hundreds of records

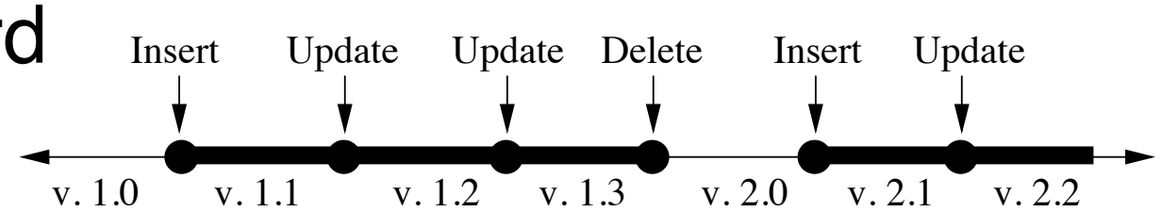
Consistency Model

- in between two extremes of general serializability and eventual consistency
- **per-record timeline consistency**
- idea: all replicas of a given record apply all updates to the record **in the same order**
- versioning of records
- replicas always move forward in time
- example for a particular primary key:



Consistency Model Implementation

- one of the replicas is designated the master
- independently for **each** record
- all updates to that record are forwarded to the master
- master replica for a record is adaptively changed to suit workload
- => replica receiving the majority of write-request for that record becomes master
- => master replica will be near the user for many apps
- record carries sequence number that is incremented on every write
- sequence number consists of generation of the record and version
- keep only one version of each record at each replica (could be changed)



API: Reading

- read-any:
  - returns a possible stale version of the record
  - departs from strict serializability: even after doing a successful write we may see an old version of the recortd
  - this call favors low latency over consistency
- read-critical(required_version):
  - returns a version of the record that is strictly newer than, or the same as the required_version
  - typical application
    - user writes a record
    - wants to read a version of the record that definitely reflects his changes
    - possible as write calls return the version number of record written

API: Reading

- read-latest:
  - returns latest copy of the record
  - reflects all writes that have succeeded
  - both read-critical and read-latest may have a higher latency than read-any
  - **why?**
  - reason: local copy may be too stale and system might need to locate a newer version at a remote replica

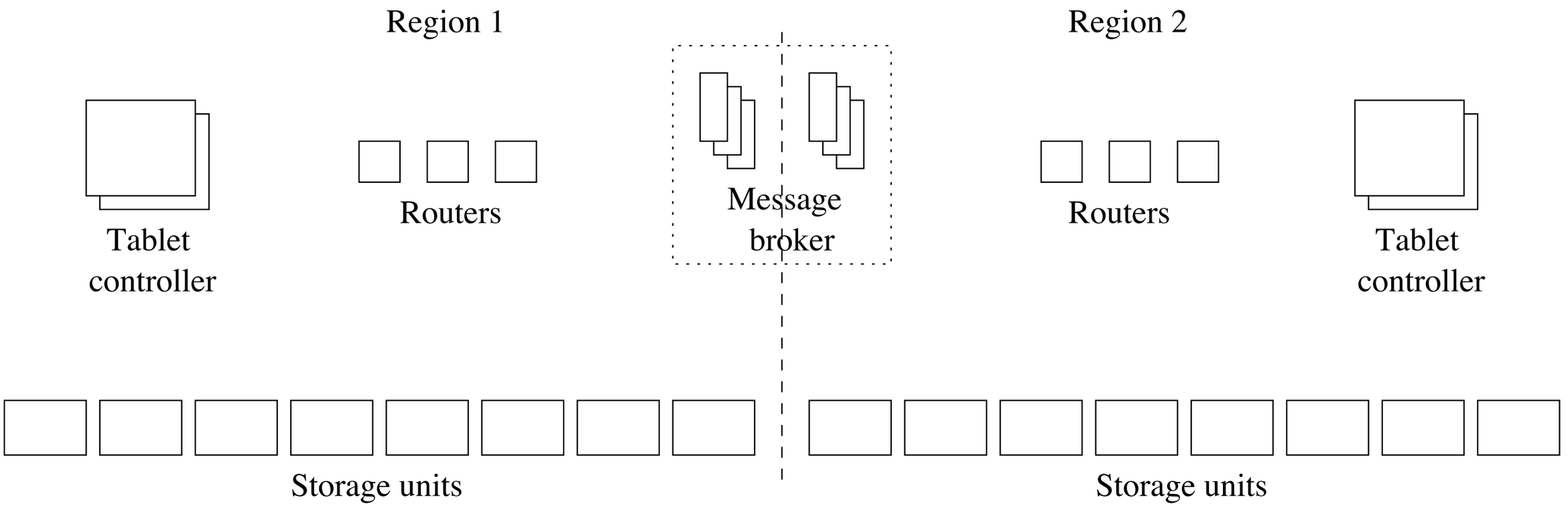
API: Writing

- write:
  - same ACID guarantees as a TA with a single write operation in it
  - useful for **blind writes** (= write without reading data before)
  - example: user updating his status on his profile
- Test-and-set-write(required_version):
  - performs the requested write to the record if and only if the present version of the record is the same as the required version
  - can be used to implement **single-row TAs** without locks
  - example: first read a record and then do a write to the record based on that read
  - use-case: incrementing a counter
  - test-and-set-write ensures that two such concurrent TAs are properly serialized
  - this is a well-known form of optimistic concurrency control

API: Discussion

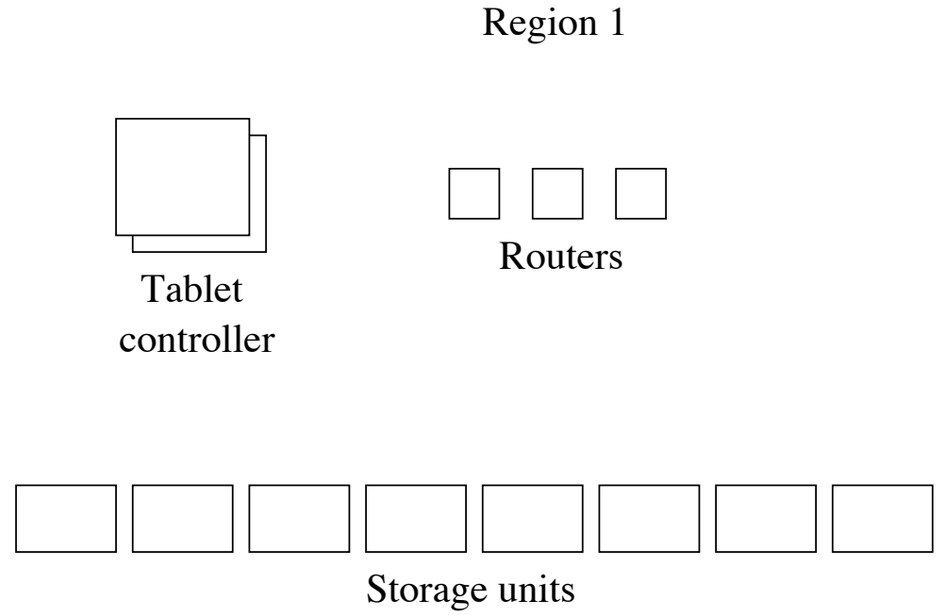
- API allows apps to indicate cases where it can do with some relaxed consistency for higher performance
- example: read-critical call
- however, no guarantees on multi-record transactions
- some extensions planned:
  - what if entire region that hosts a master copy becomes unreachable?
  - may need to branch timeline
  - need automatic conflict resolution and notifications
  - discarding and merging timeline branches

System Architecture



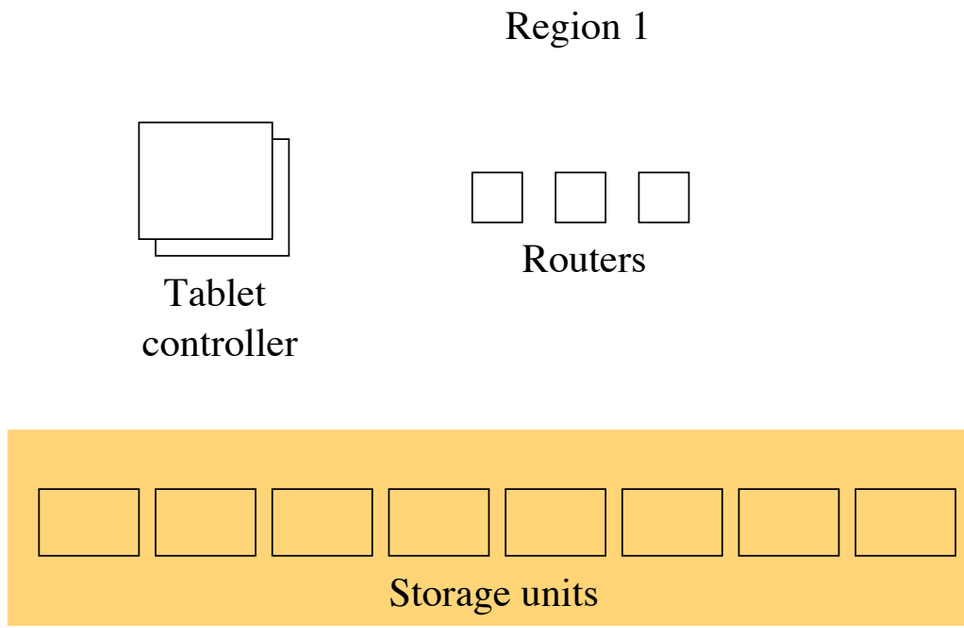
- system divided into regions
- each region contains full complement of system components plus complete copy of each table
- regions typically geographically distributed
- **no** traditional database log (well, we will see...)
- pub/sub mechanism provides redo

System Architecture: Inside a Region



- tables are horizontally partitioned
- a partition is termed a **tablet**
- tablets scattered across servers
- server might have hundreds of thousands of tablets
- typical tablet has about few hundred or a few gigabytes
- typical tablet contains tens or thousands of records
- assignment of tablets to servers flexible
- good for load balancing
- allows for easy rebalancing
- examples:
  - server gets overloaded => move some tablets to other machine
  - server crashes => spread recovered tablets to working machines

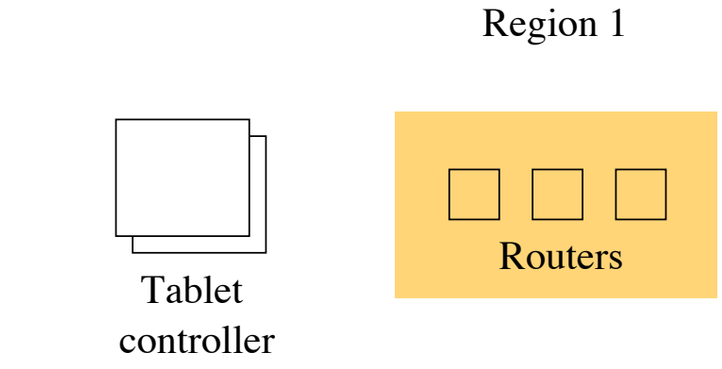
Storage Units



- store tablets
- respond to get(), scan(), and update()
- may use any physical storage layer that seems appropriate
- hash tables: UNIX file-system based hash table
- ordered tables: use MySQL with InnoDB (optimized for updates)
- records stored as parsed JSON objects
- JavaScript Object Notation
- needs less space than XML

```
{
      "firstName": "John",
      "lastName": "Smith",
      "address": {
          "streetAddress": "21 2nd Street",
          "city": "New York",
          "state": "NY",
          "postalCode": 10021
      },
      "phoneNumbers": [
          "212 555-1234",
          "646 555-4567"
      ]
  }
```

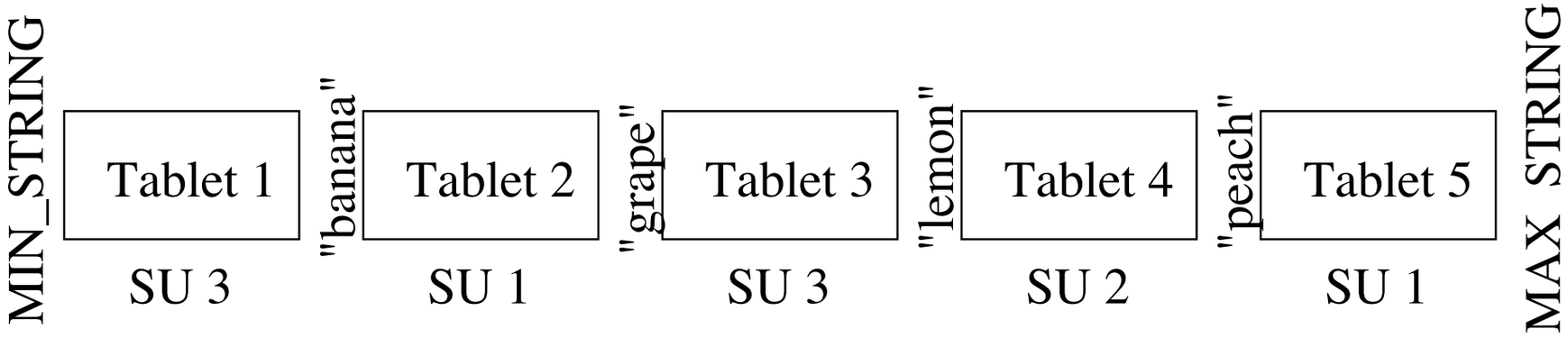
Routers



- determines which storage unit is responsible for a given record
  - which tablet contains the record?
  - which storage unit contains that tablet?
- both for reads and writes
- for ordered tables primary-key space is divided into intervals
- each interval corresponds to one tablet

why?

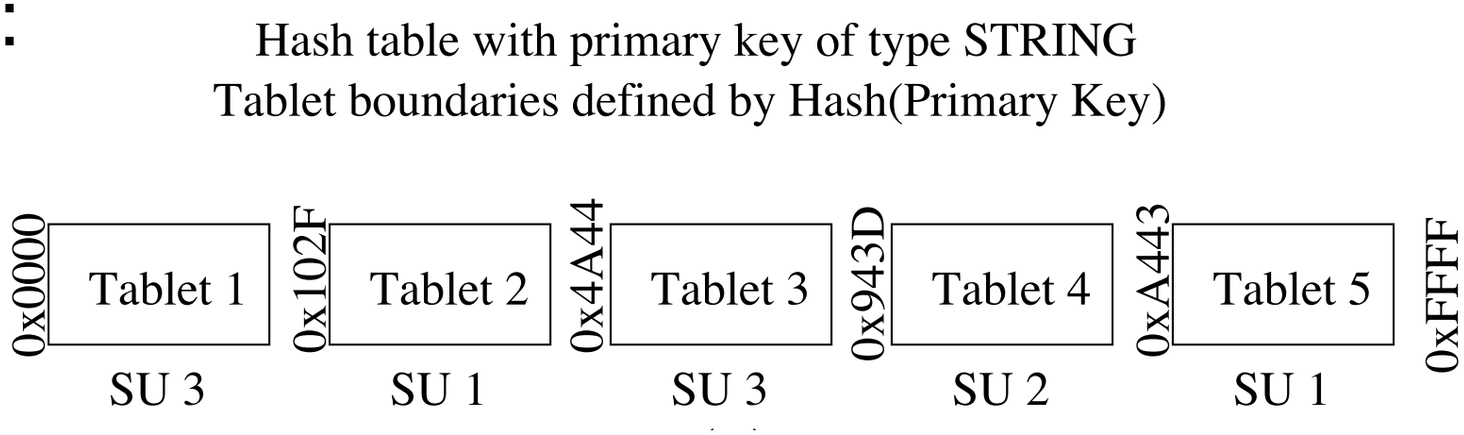
- example mapping of interval range to tablets:



- binary search on intervals to find tablet

Routers

- hash functions produce hash in RANGE 0 <= H() < 2^n
- hash space [0...2^n] divided into intervals
- each interval corresponds to a single tablet
- example:



- first apply hash function
- then again binary search on intervals to find tablet
- system does **not** use linear or extensible hashing
- as method very similar to interval range search (same code)

Routers

- interval mappings are kept in main memory
- inexpensive to search
- planned scale:
  - 100 servers per region
  - 100 tablets each
  - assume keys are 100 bytes (high end of apps)
  - => mapping will take a few hundred MB of RAM on a server
  - tablet contains about 500 MB
  - => DB of about 500 Terabytes
  - for larger mappings may need disk-based mapping
- remember the discussion we had on how to map record IDs to pages?

Slide 13 from Week2:

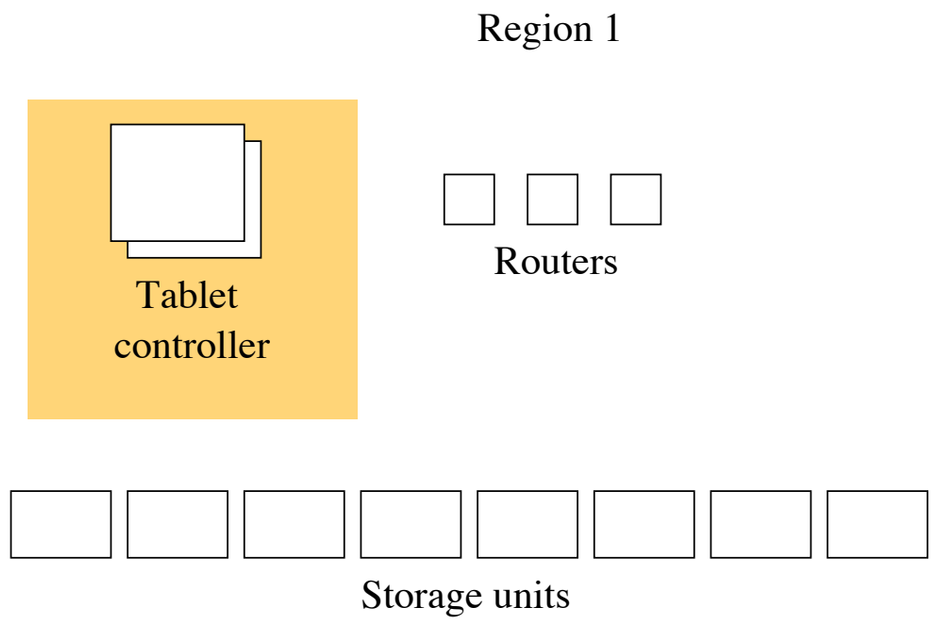
SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

Indirect Addressing: Mapping Table

- Idea:
  - 1. keep a separate mapping table
  - 2. hide physical addresses
    (outside world only knows logical addresses)
  - no forwarding
  - if tuple needs to be moved: change entry in mapping table

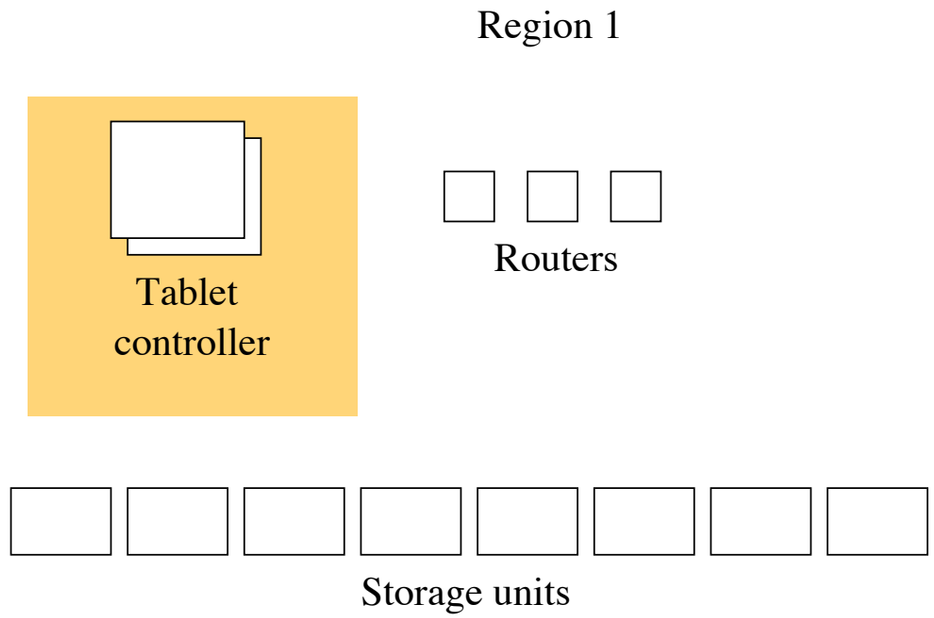
- In the context of PNUTS:
  - router = mapping table
  - move from tablet to other tablet
    = move tuple from one page to another

Tablet Controllers



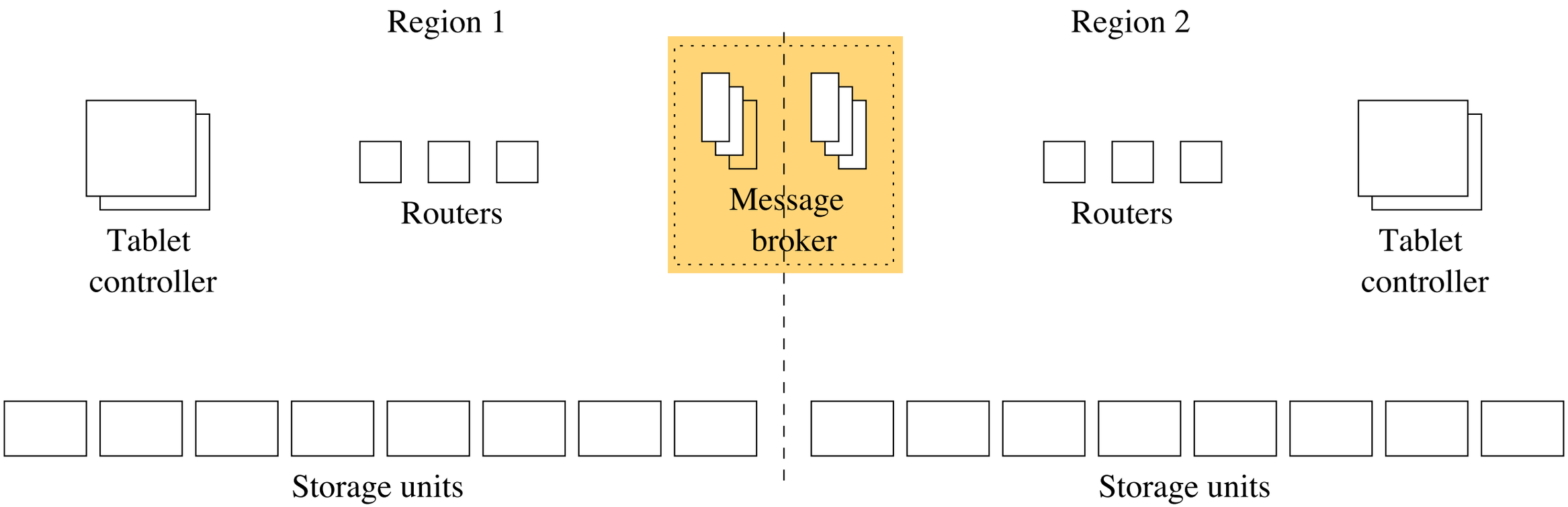
- owns the master copy of the mapping
- routers only get replicas
- routers need to poll the tablet controller periodically to receive update
- tablet controller decides on tablet-> storage unit mapping
  - load balancing
  - recovery
  - large tablet must be split
- if tablet controller updates mappings, router mapping may be out of date for a short period of time
- => requests will be misdirected
- misdirected request to a storage unit triggers error at st. unit
- => forces router to get a new copy from the tablet controller

Tablet Controllers



- if a router fails, simply start a new one
- no recovery for router needed
- new router retrieves copy of the mapping from tablet controller
- in addition: tablet controller kept redundantly
  - actually a pair of tablet controllers:
    - one active
    - one hot standby

Message Brokers



- **message broker**: pub/sub system
- log replacement
- an update is considered committed if and only if the update has been published to the message broker
- Note: this is WAL replacing the log by a message broker

Message Brokers

- logs messages on multiple disks on multiple servers
- messages are not purged from a broker until PNUTS has applied the update to all replicas
- messages sent to one broker will be distributed to other brokers
- these brokers then send messages to the routers and tablets

Consistency via YMB and Mastership

- problem: messages published to different brokers may be delivered in any order
- thus does not provide timeline consistency
- therefore use **per-record mastership**
- idea:
  - one copy of a record is considered the master
  - all updates are directed to that master copy
  - updates are propagated by publishing them to the message broker
  - master record publishes updates to a **single** broker (in order)
  - thus updates are delivered to replicas in commit order
  - once the update is published, we treat it as committed

Consistency via YMB and Mastership

- updates for records may originate in a non-master region
- these updates must be forwarded to the master replica before being committed
- each record also keeps in a hidden metadata field identity of its current master
- this field is checked whenever an update arrives
- mastership of a record may migrate
- example: user moves from Wisconsin to California
- same ideas used for tablets in order to enforce primary key constraints when inserting records
  - one tablet is the **tablet master**
  - tablet master decides on order of inserts

Recovery

- how to recover in case of a storage unit failure?
- simply copy lost tablet from another replica
- three steps
  1. tablet controller requests copy from a remote replica termed the "source tablet"
  2. a checkpoint message is published to YMB ensures that updates that are coming in while copying are later applied to the tablets
  3. source tablet is copied to the destination region
- to make this more efficient, PNUTS keeps tablet boundaries in synch across regions!
- tablet splits are performed by having all regions split the tablet at the same time
- uses two-phase-commit (2PC) between regions

Hosted Database Service

- PNUTS is hosted
- centrally managed
- multiple apps run on this platform
- apps may have different workloads
- needs performance isolation:
  - one heavyweight application should not have an impact on other applications
  - currently handled by assigning diferent apps to different sets of storage units within a region

PNUTS Applications: User Database

- hundreds of millions of active IDs
- billions of total IDs
- contains user preferences
- profile information, usage statistics, etc.
- record timeline model a good fit:
  - user should see his/her own changes immediately
  - others do not need to see it immediately
- also useful in the hosted model
- => different Yahoo! apps may share the user database

Conclusions

- PNUTS: globally distributed database system
- nice re-combination of existing DBMS-technology
- redundancy at multiple levels to support recovery
- extremely scalable
- flexible addition removal of resources due to interval mapping
- per-record timeline consistency
- in-between eventual consistency and serializable

Next Topic: map/reduce.