UNIVERSITY OF WATERLOO

Data-Intensive Distributed Computing

CS 431/631 451/651 (Fall 2019)

Part 7: Mutable State (1/2)

November 12, 2019

Ali Abedi

These slides are available at https://www.student.cs.uwaterloo.ca/~cs451

Structure of the Course

Analyzing Text

Analyzing Graphs

Analyzing Relational Data

Data Mining

"Core" framework features and algorithm design

The Fundamental Problem

We want to keep track of *mutable* state in a *scalable* manner

Assumptions:

State organized in terms of logical records

State unlikely to fit on single machine, must be distributed

MapReduce won't do!

Want more? Take a *real* distributed systems course!

The Fundamental Problem

We want to keep track of *mutable* state in a *scalable* manner

Assumptions:

State organized in terms of logical records

State unlikely to fit on single machine, must be distributed

Uh… just use an RDBMS?

What do RDBMSes provide?

Relational model with schemas

Powerful, flexible query language

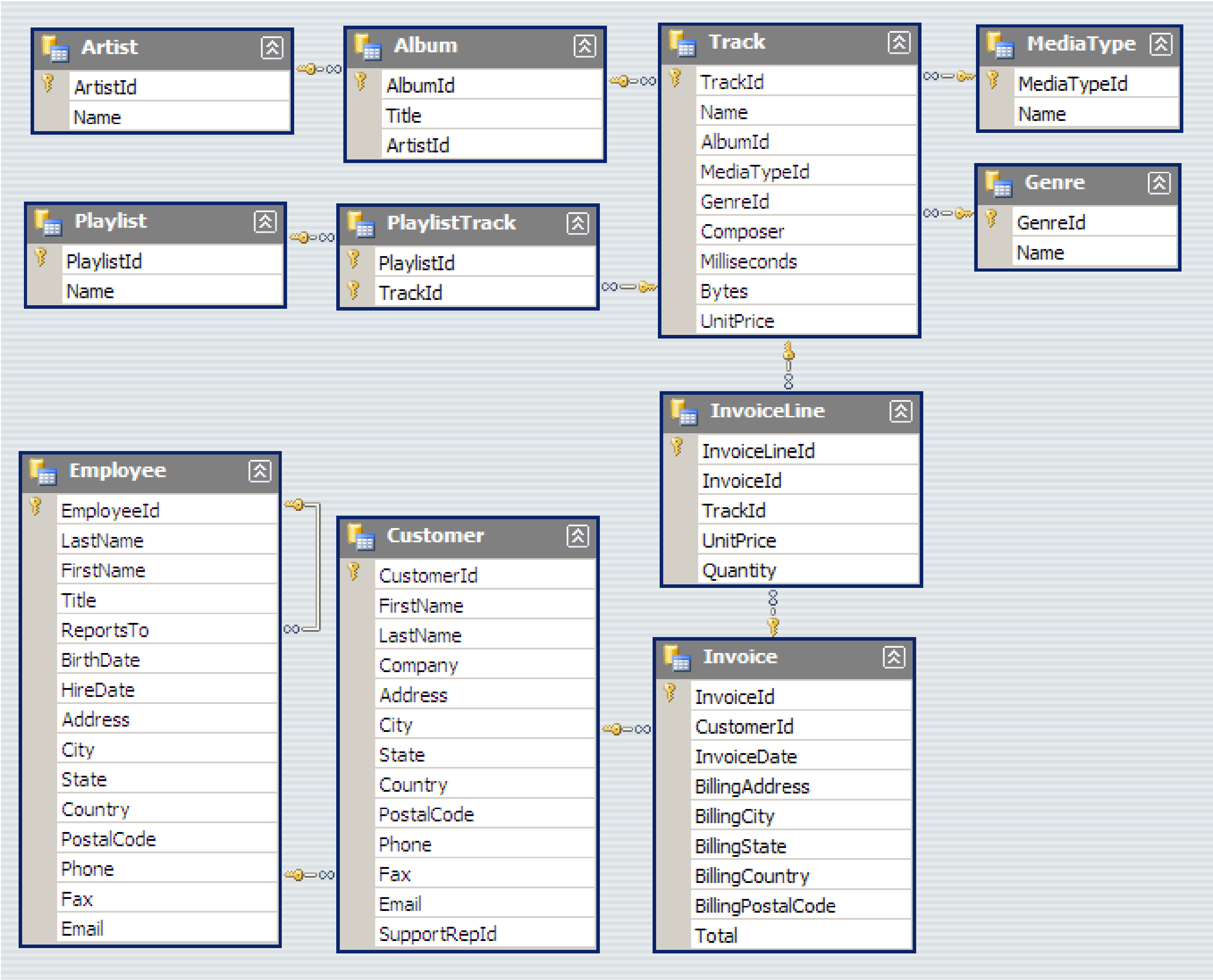
Transactional semantics: ACID

Rich ecosystem, lots of tool support

RDBMSes: Pain Points

Source: www.flickr.com/photos/spencerdahl/6075142688/

#1: Must design up front, painful to evolve



#2: Pay for ACID!

Source: Wikipedia (Tortoise)

#3: Cost!

Source: www.flickr.com/photos/gnusinn/3080378658/

What do RDBMSes provide?

Relational model with

Powerful, flexible quer

Transactional seman

Rich ecosystem, lots of

What if we want *a la carte*?

Source: www.flickr.com/photos/vidiot/18556565/

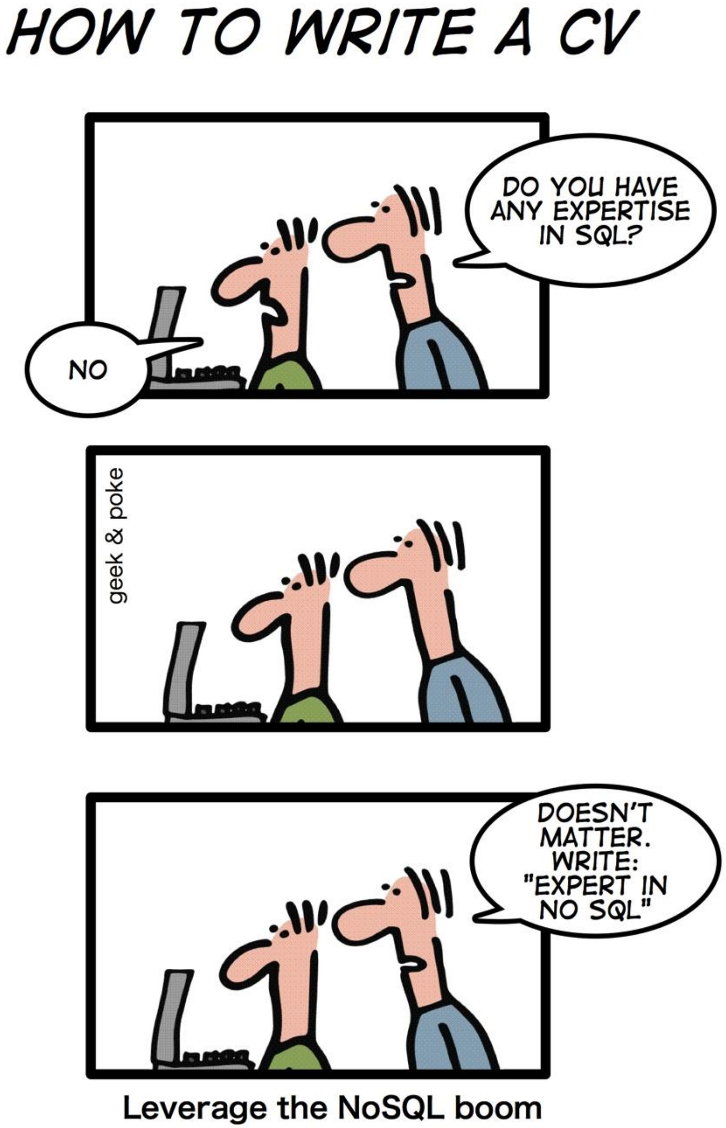
Features *a la carte*?

What if I'm willing to give up consistency for scalability?

What if I'm willing to give up the relational model for flexibility?

What if I just want a cheaper solution?

Enter… NoSQL!



Source: geekandpoke.typepad.com/geekandpoke/2011/01/nosql.html

NoSQL
(Not only SQL)

1. Horizontally scale "simple operations"
2. Replicate/distribute data over many servers
3. Simple call interface
4. Weaker concurrency model than ACID
5. Efficient use of distributed indexes and RAM
6. Flexible schemas

But, don't blindly follow the hype…
Often, MySQL is what you really need!

Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*.

“web scale”

(Major) Types of NoSQL databases

Key-value stores

Column-oriented databases

Document stores

Graph databases

Three Core Ideas

Keeping track of the partitions?

Partitioning (sharding)

To increase scalability and to decrease latency

Consistency?

Replication

To increase robustness (availability) and to increase throughput

Consistency?

Caching

To reduce latency

Key-Value Stores

Source: Wikipedia (Keychain)

Key-Value Stores: Data Model

Stores associations between keys and values

Keys are usually primitives

For example, ints, strings, raw bytes, etc.

Values can be primitive or complex: often opaque to store

Primitives: ints, strings, etc.

Complex: JSON, HTML fragments, etc.

Key-Value Stores: Operations

Very simple API:

Get – fetch value associated with key

Put – set value associated with key

Optional operations:

Multi-get

Multi-put

Range queries

Secondary index lookups

Consistency model:

Atomic single-record operations (usually)

Cross-key operations: who knows?

Key-Value Stores: Implementation

Non-persistent:

Just a big in-memory hash table

Examples: Redis, memcached

Persistent

Wrapper around a traditional RDBMS

Examples: Voldemort

What if data doesn't fit on a single machine?

Simple Solution: Partition!

Partition the key space across multiple machines

Let's say, hash partitioning

For n machines, store key *k* at machine *h(k) mod n*

Okay… But:

How do we know which physical machine to contact?

How do we add a new machine to the cluster?

What happens if a machine fails?

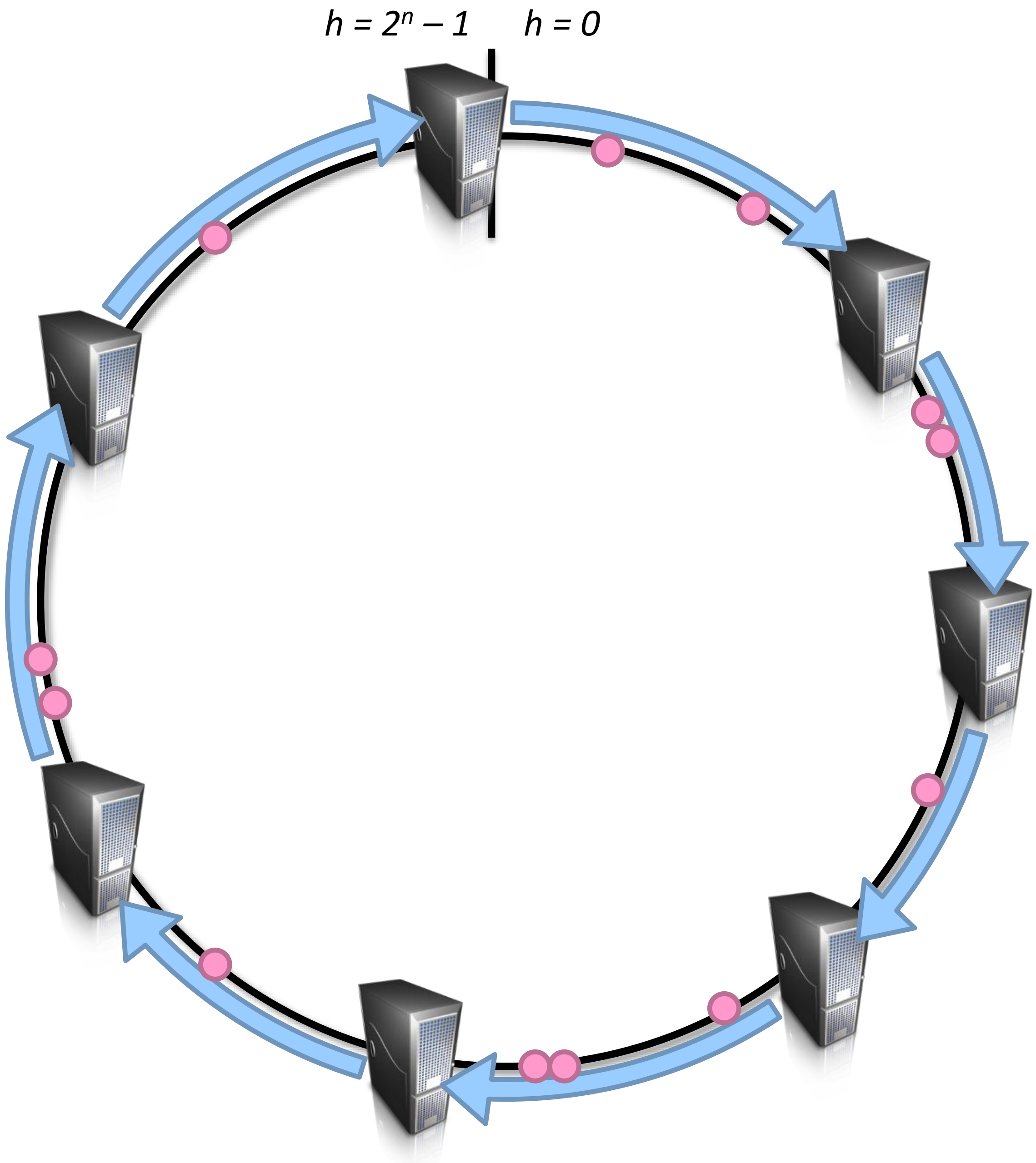
Clever Solution

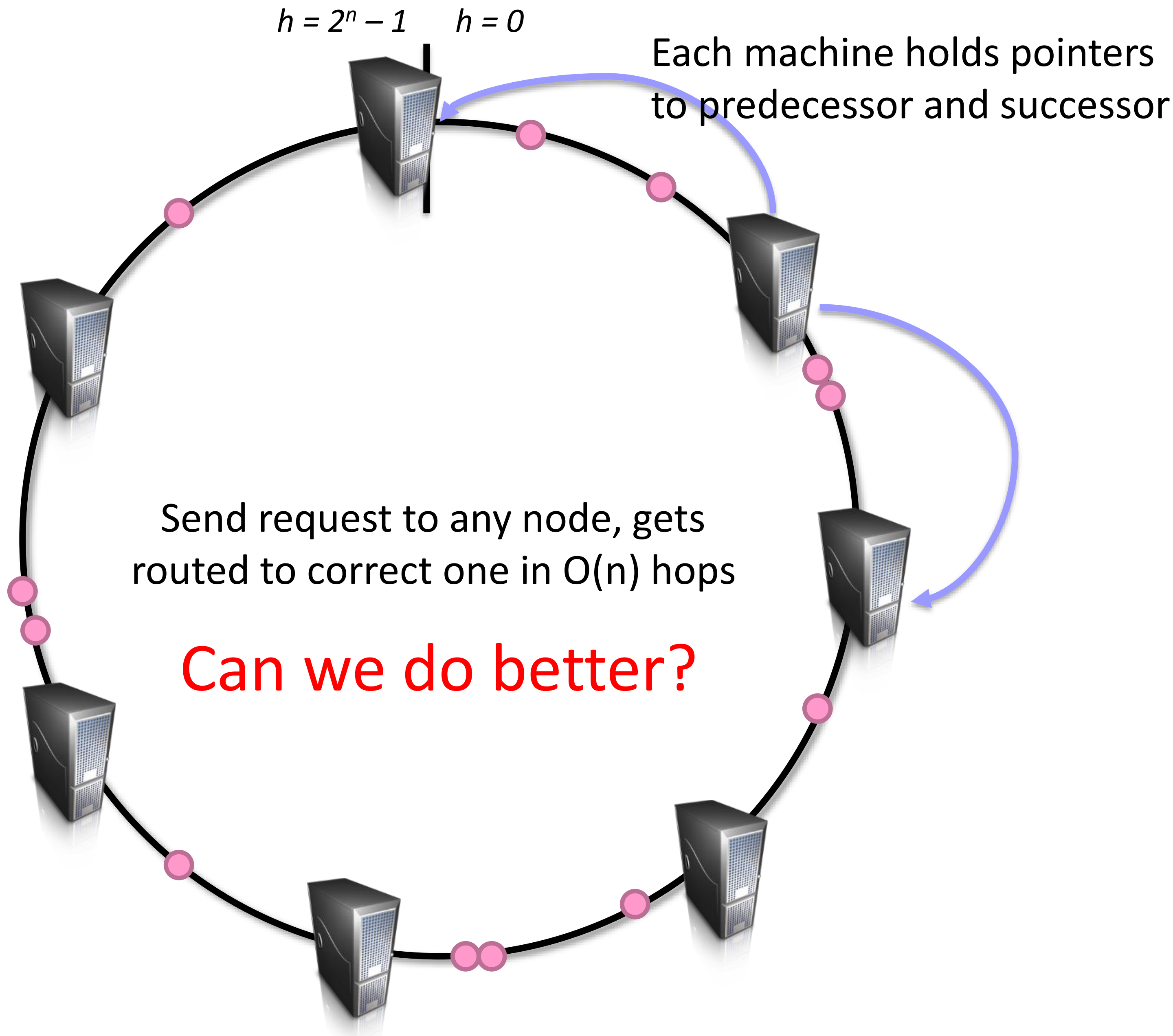
Hash the keys

Hash the machines also!

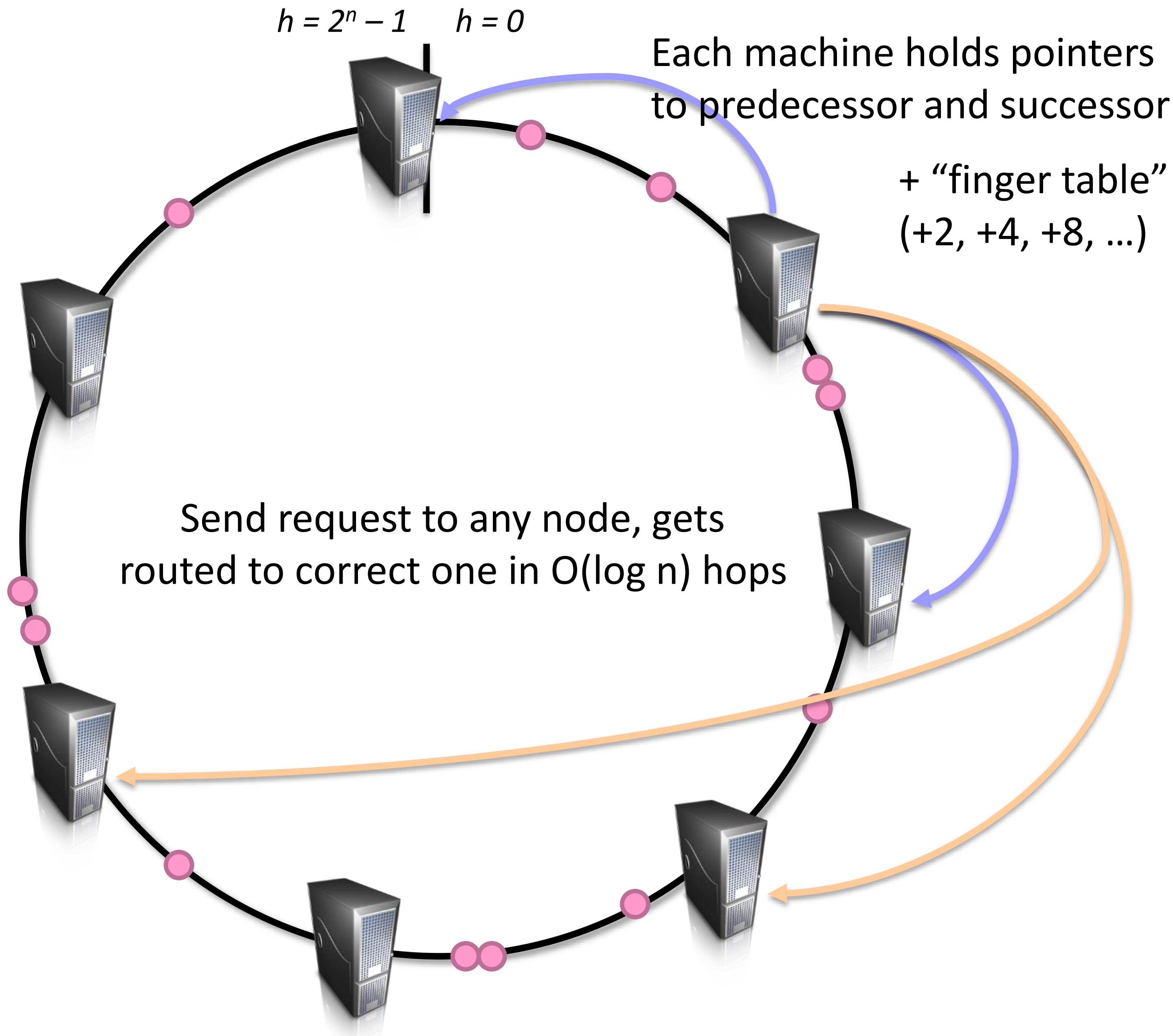
Distributed hash tables!

(following combines ideas from several sources…)


h = 2n – 1
h = 0

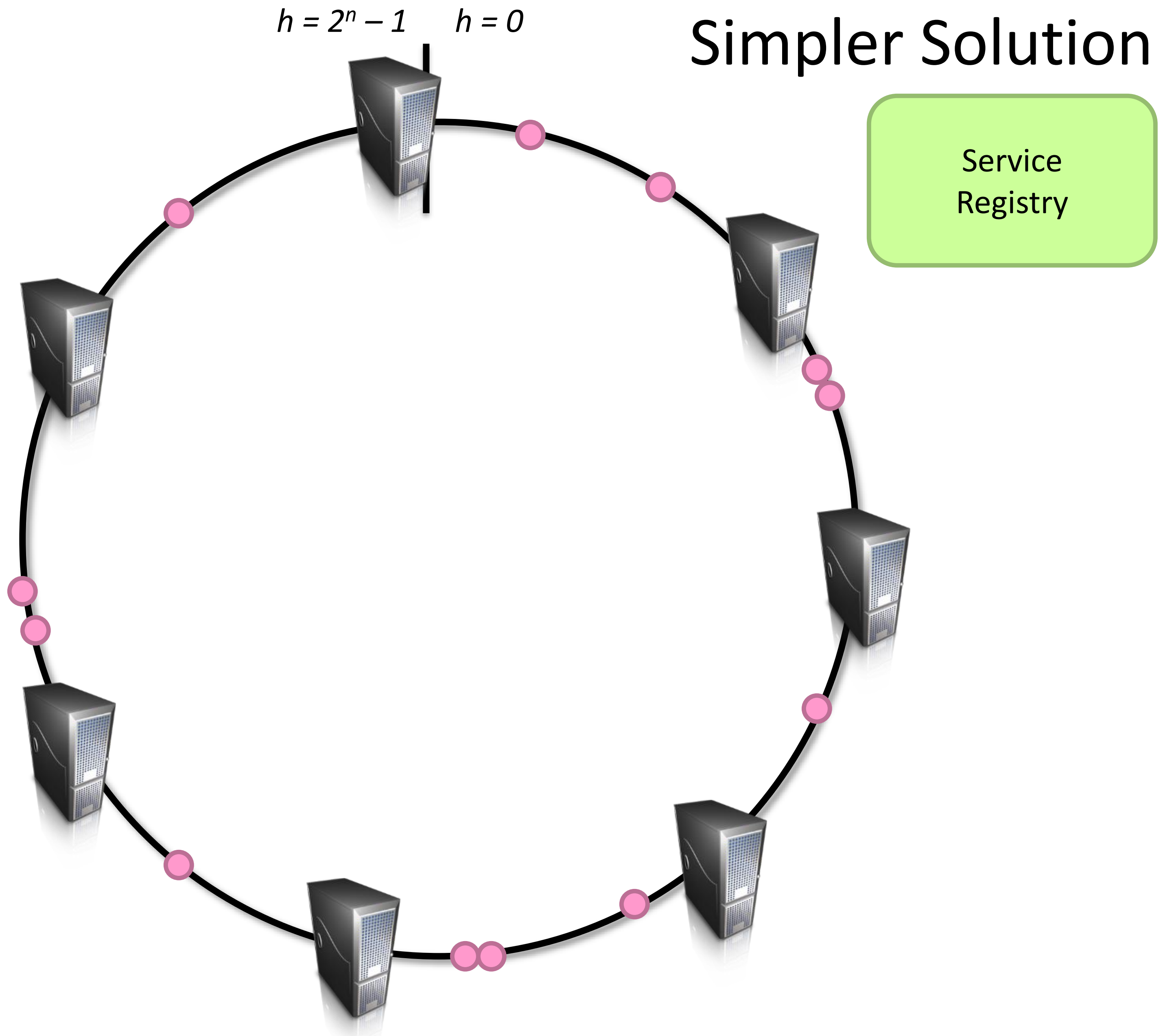


Routing: Which machine holds the key?



Routing: Which machine holds the key?

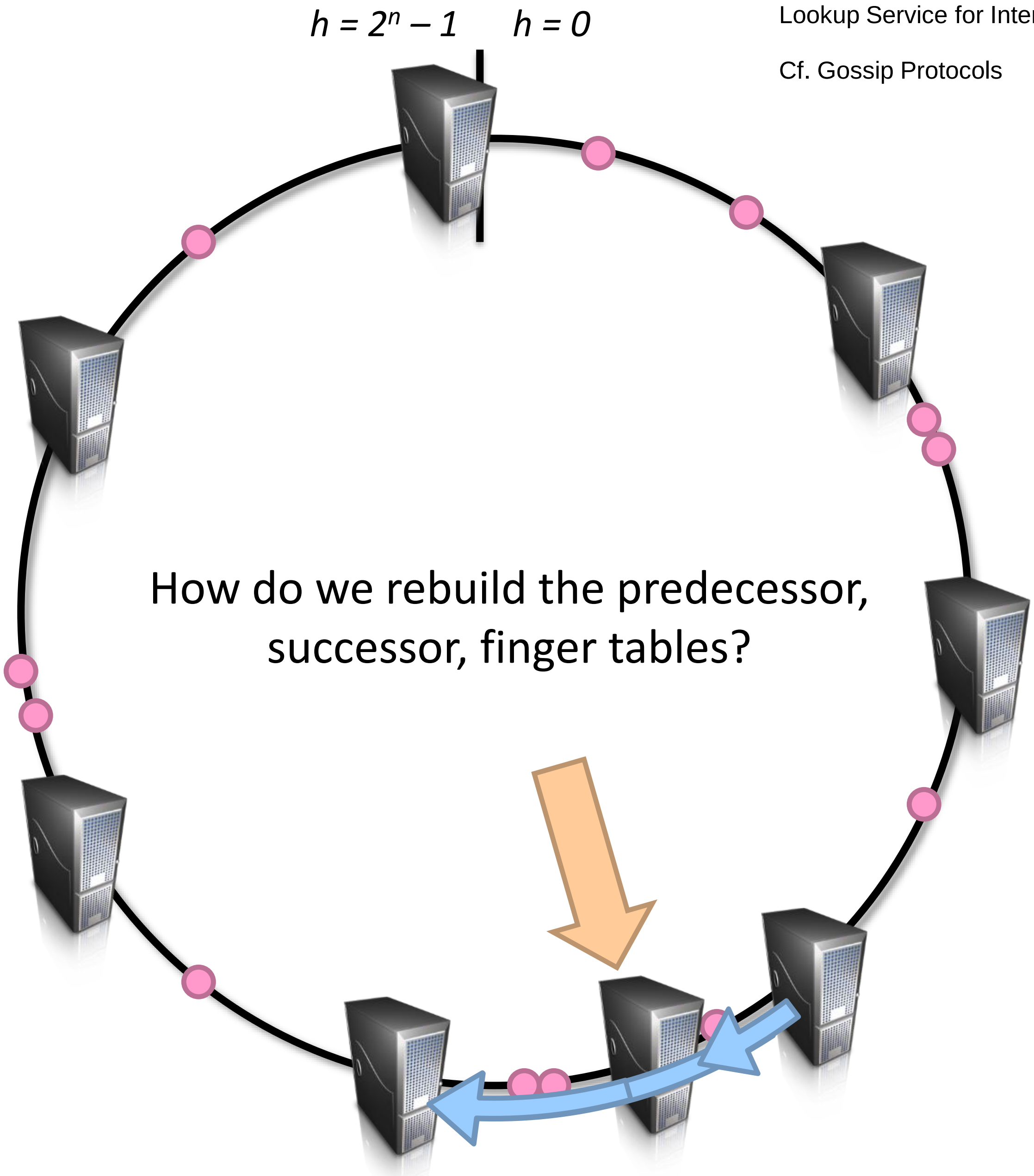
Simpler Solution



Routing: Which machine holds the key?

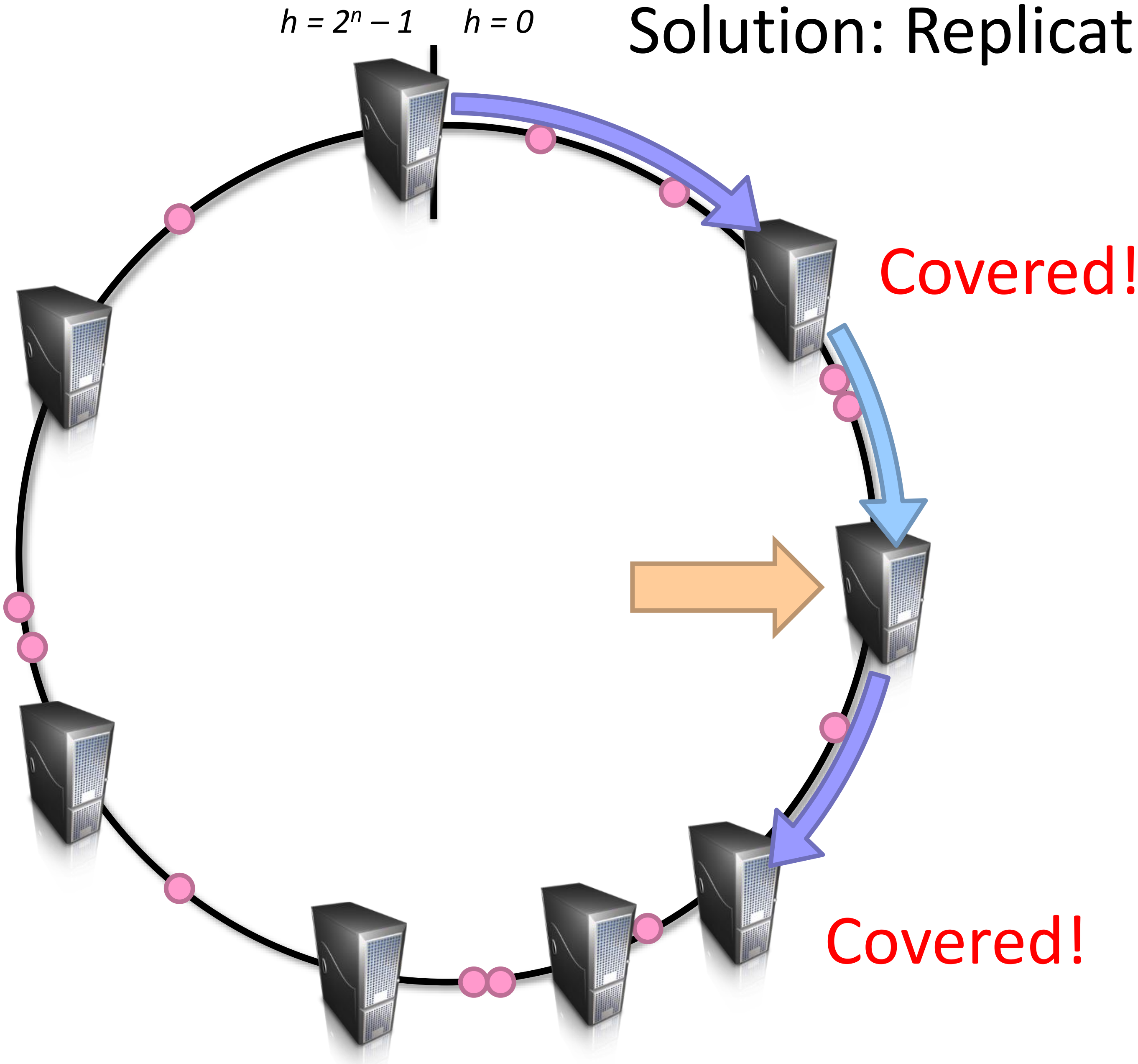
Stoica et al. (2001). Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *SIGCOMM.*

Cf. Gossip Protocols



New machine joins: What happens?

Solution: Replication



Machine fails: What happens?

Three Core Ideas

Keeping track of the partitions?

Partitioning (sharding)

To increase scalability and to decrease latency

Consistency?

Replication

To increase robustness (availability) and to increase throughput

Caching

To reduce latency

Another Refinement: Virtual Nodes

Don't directly hash servers

Create a large number of virtual nodes, map to physical servers

Better load redistribution in event of machine failure

When new server joins, evenly shed load from other servers

Bigtable

Source: Wikipedia (Table)

Bigtable Applications

Gmail

Google's web crawl

Google Earth

Google Analytics

Data source and data sink for MapReduce

HBase is the open-source implementation…

Data Model

A table in Bigtable is a sparse, distributed, persistent multidimensional sorted map

Map indexed by a row key, column key, and a timestamp

(row:string, column:string, time:int64) → uninterpreted byte array

Supports lookups, inserts, deletes

Single row transactions only

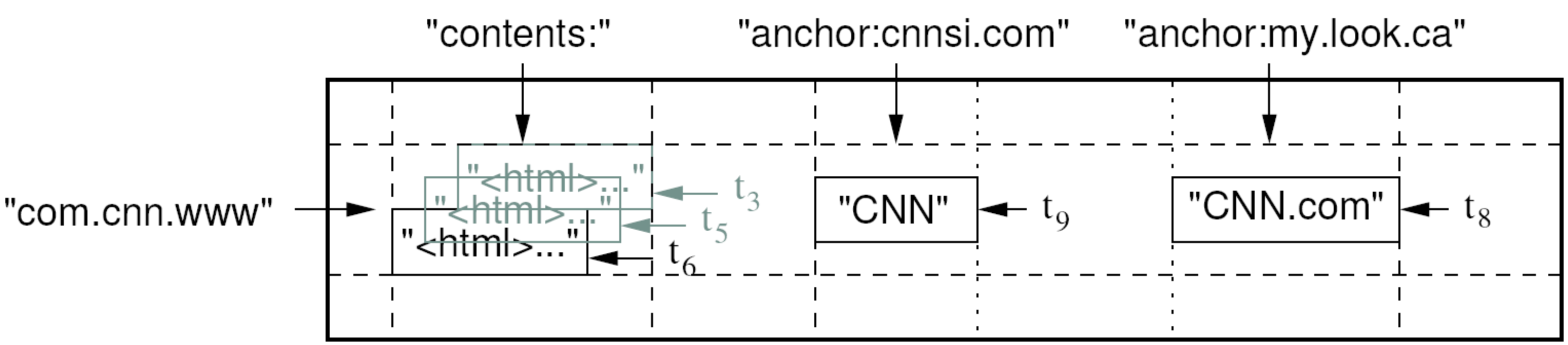


Image Source: Chang et al., OSDI 2006

Rows and Columns

Rows maintained in sorted lexicographic order

Applications can exploit this property for efficient row scans

Row ranges dynamically partitioned into tablets

Columns grouped into column families

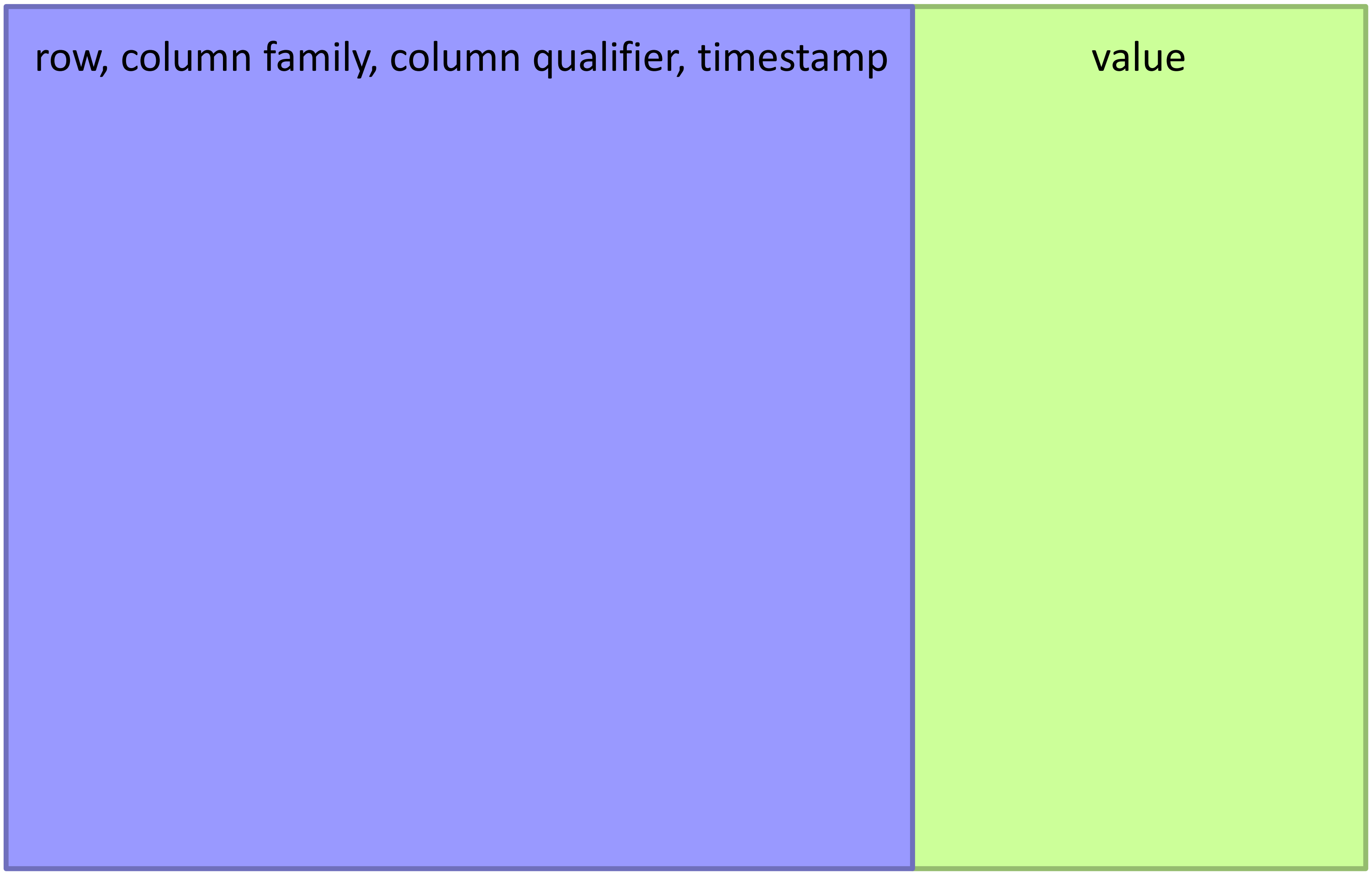
Column key = family:qualifier

Column families provide locality hints

Unbounded number of columns

At the end of the day, it's all key-value pairs!

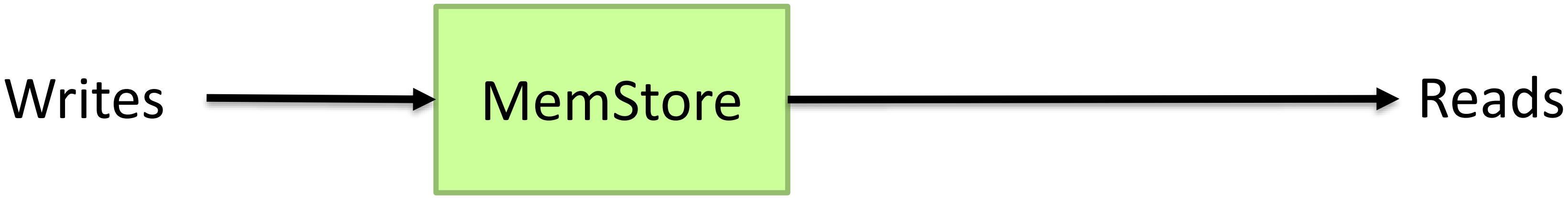
Key-Values



Okay, so how do we build it?

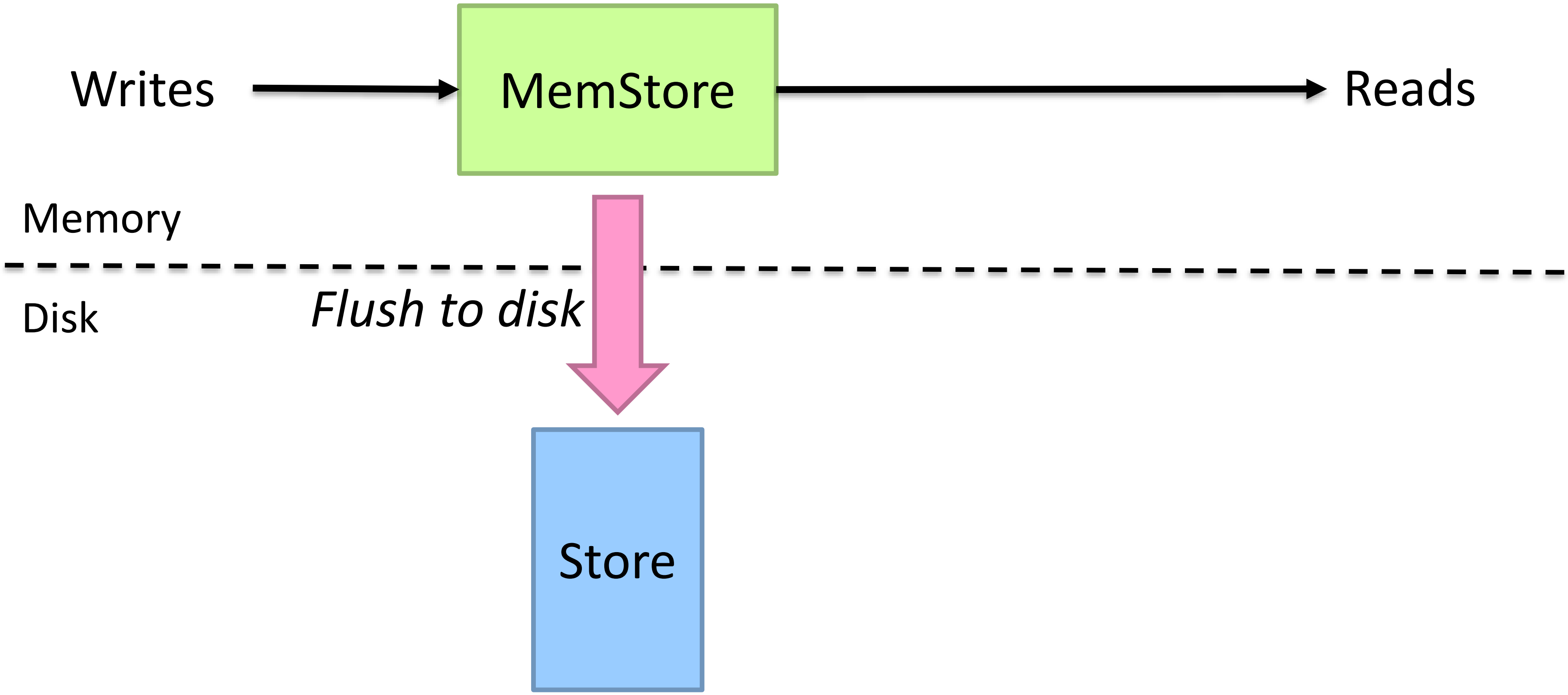
In Memory	On Disk
Mutability Easy	Mutability Hard
Small	Big

Log Structured Merge Trees



What happens when we run out of memory?

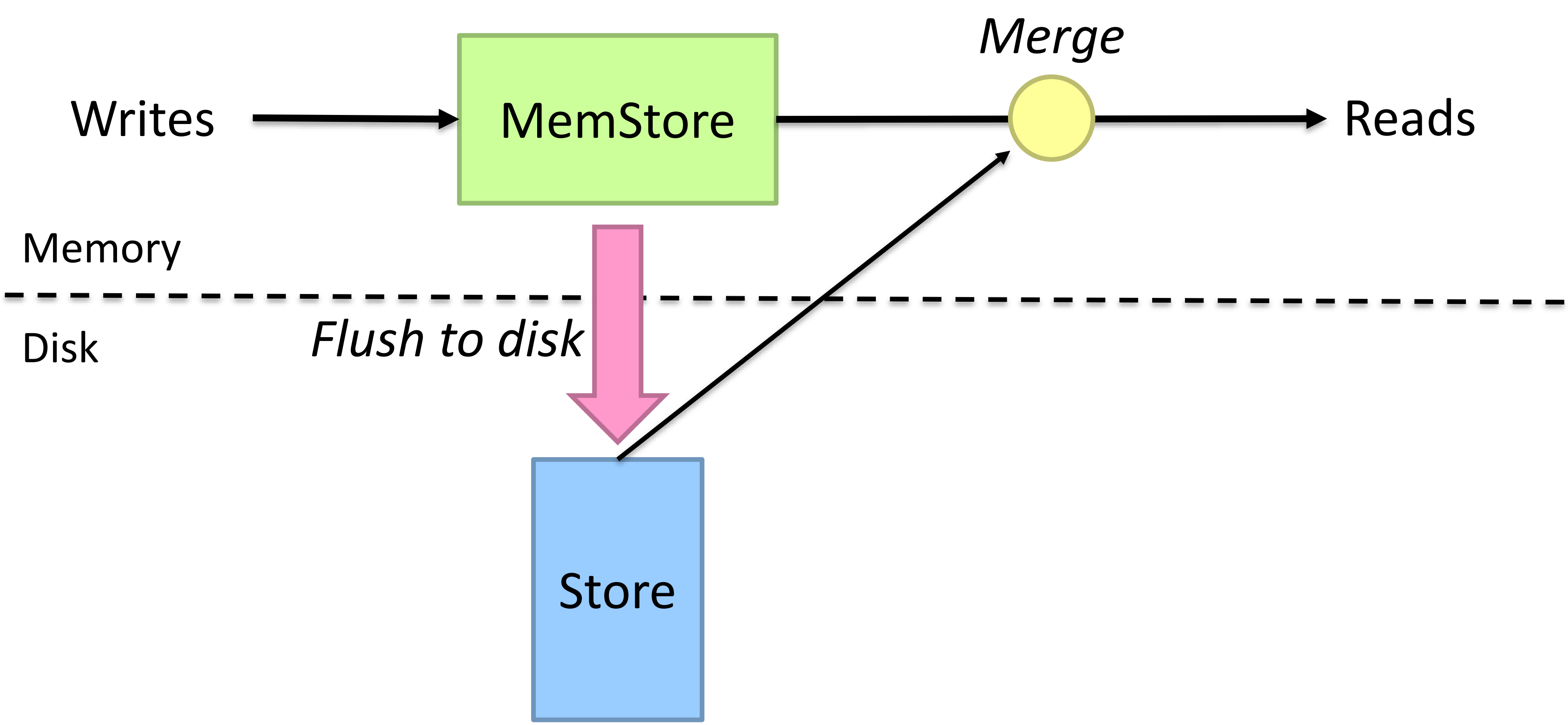
Log Structured Merge Trees



Immutable, indexed, persistent, key-value pairs

What happens to the read path?

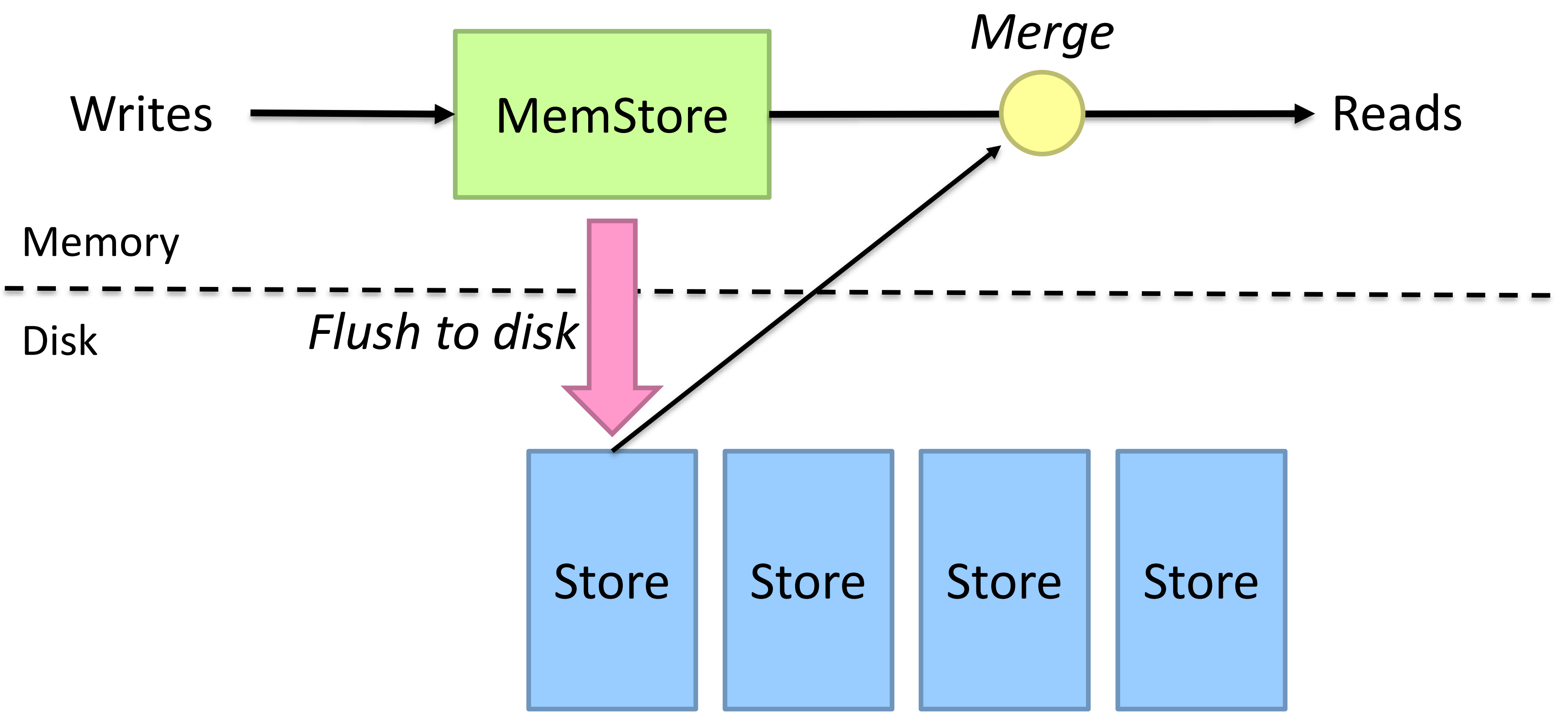
Log Structured Merge Trees



Immutable, indexed, persistent, key-value pairs

What happens as more writes happen?

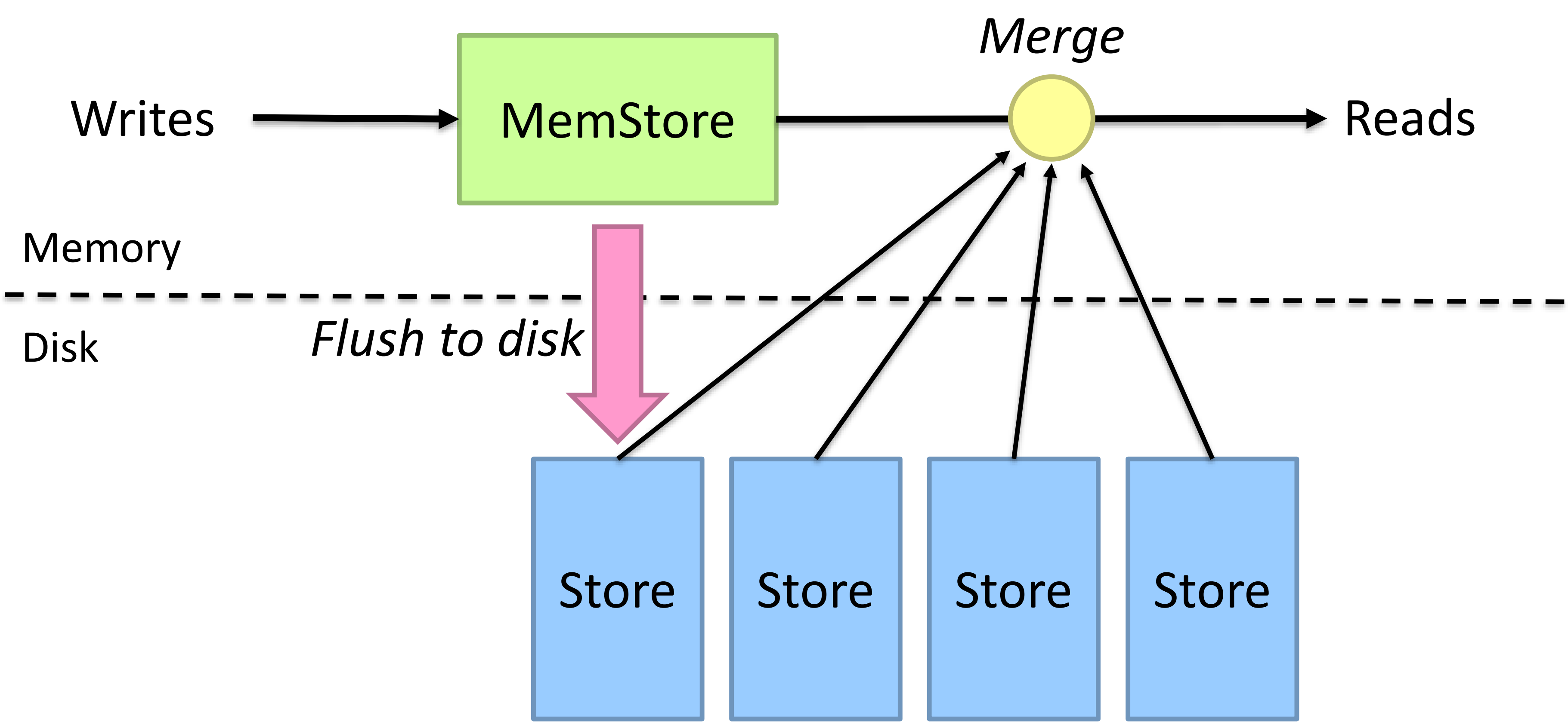
Log Structured Merge Trees



Immutable, indexed, persistent, key-value pairs

What happens to the read path?

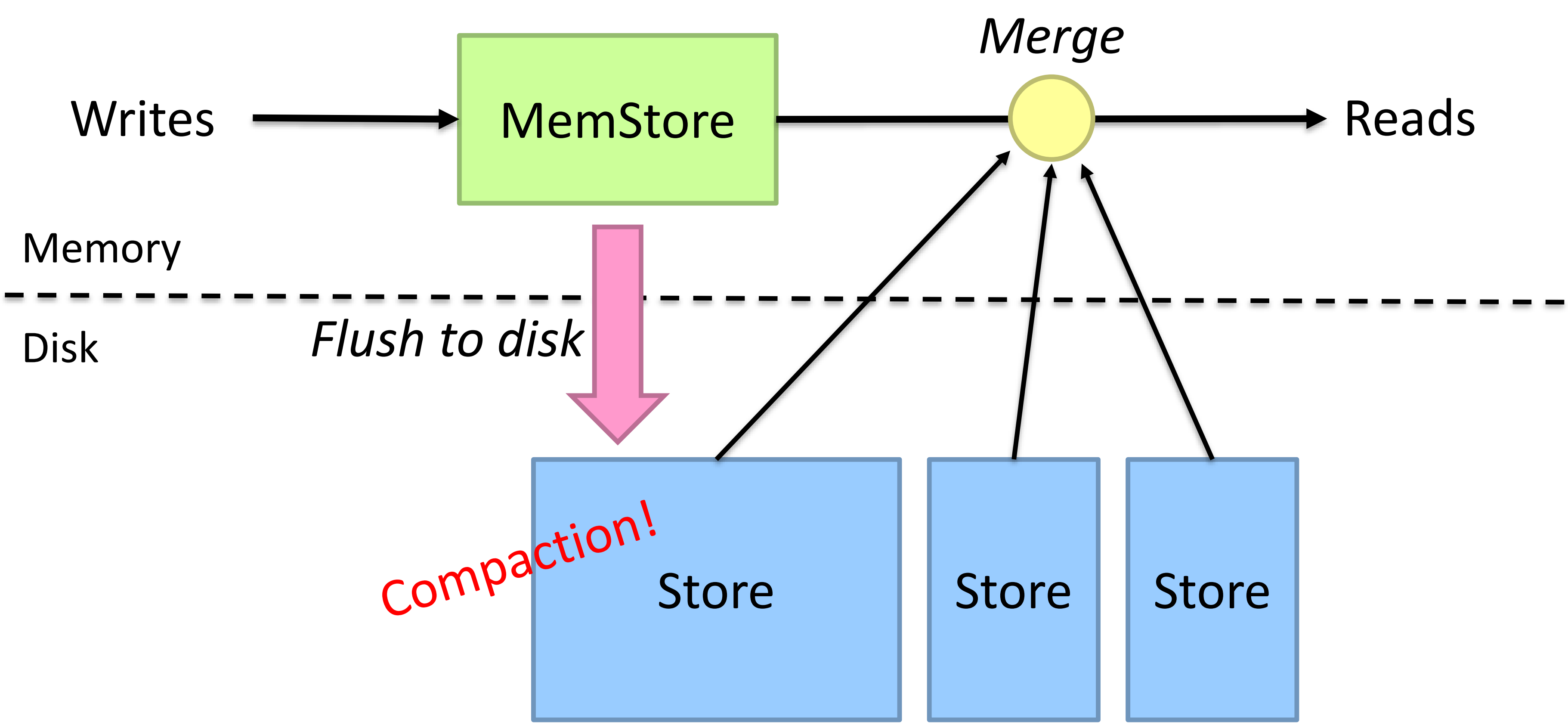
Log Structured Merge Trees



Immutable, indexed, persistent, key-value pairs

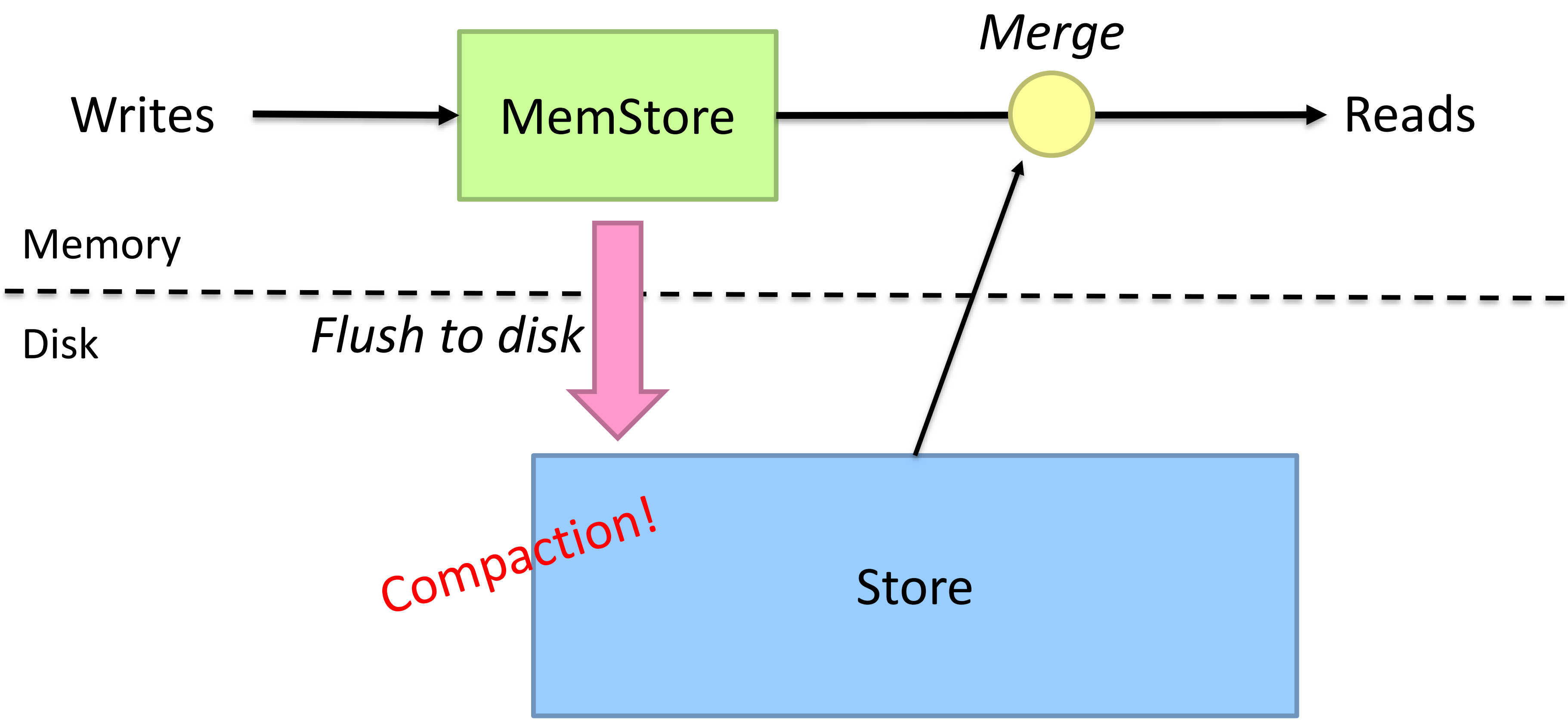
What's the next issue?

Log Structured Merge Trees



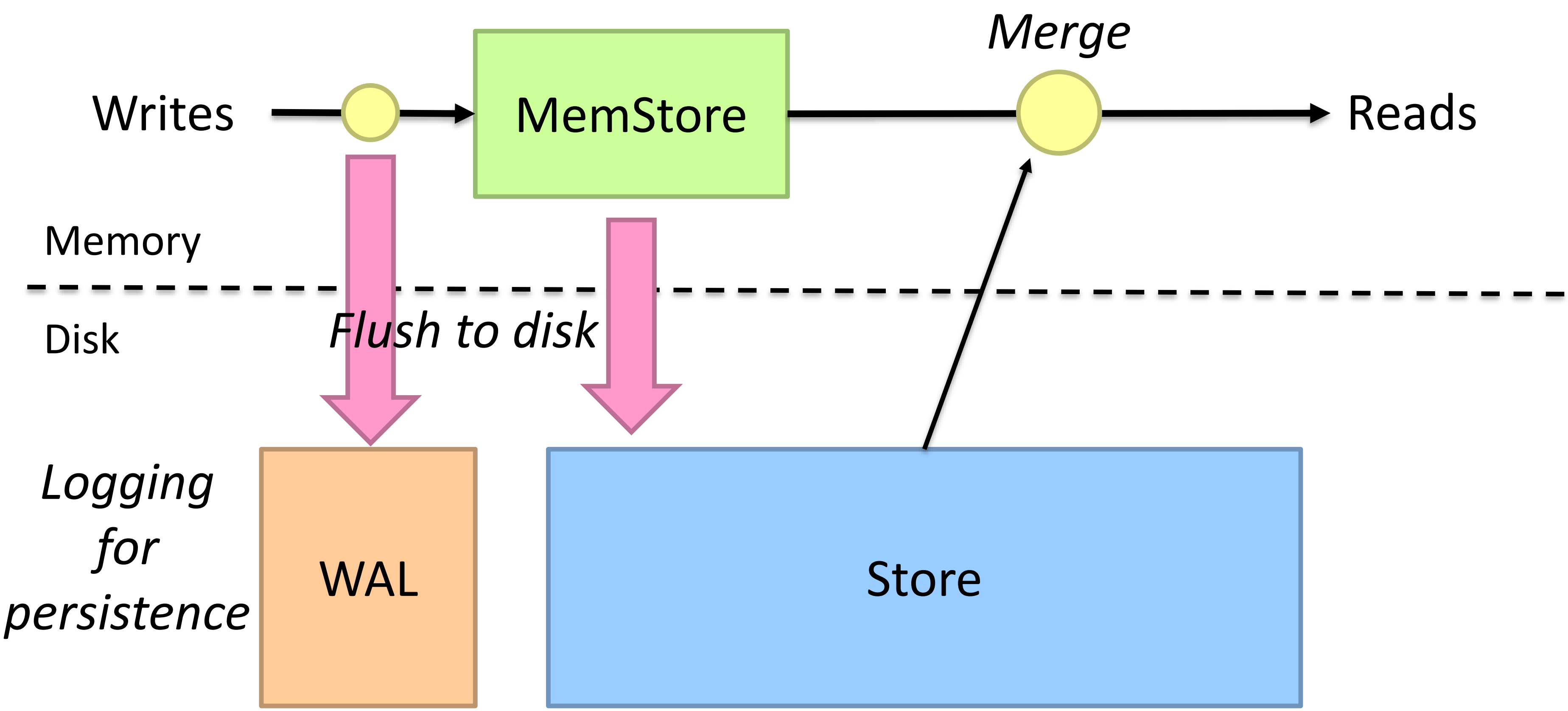
Immutable, indexed, persistent, key-value pairs

Log Structured Merge Trees



Immutable, indexed, persistent, key-value pairs

Log Structured Merge Trees

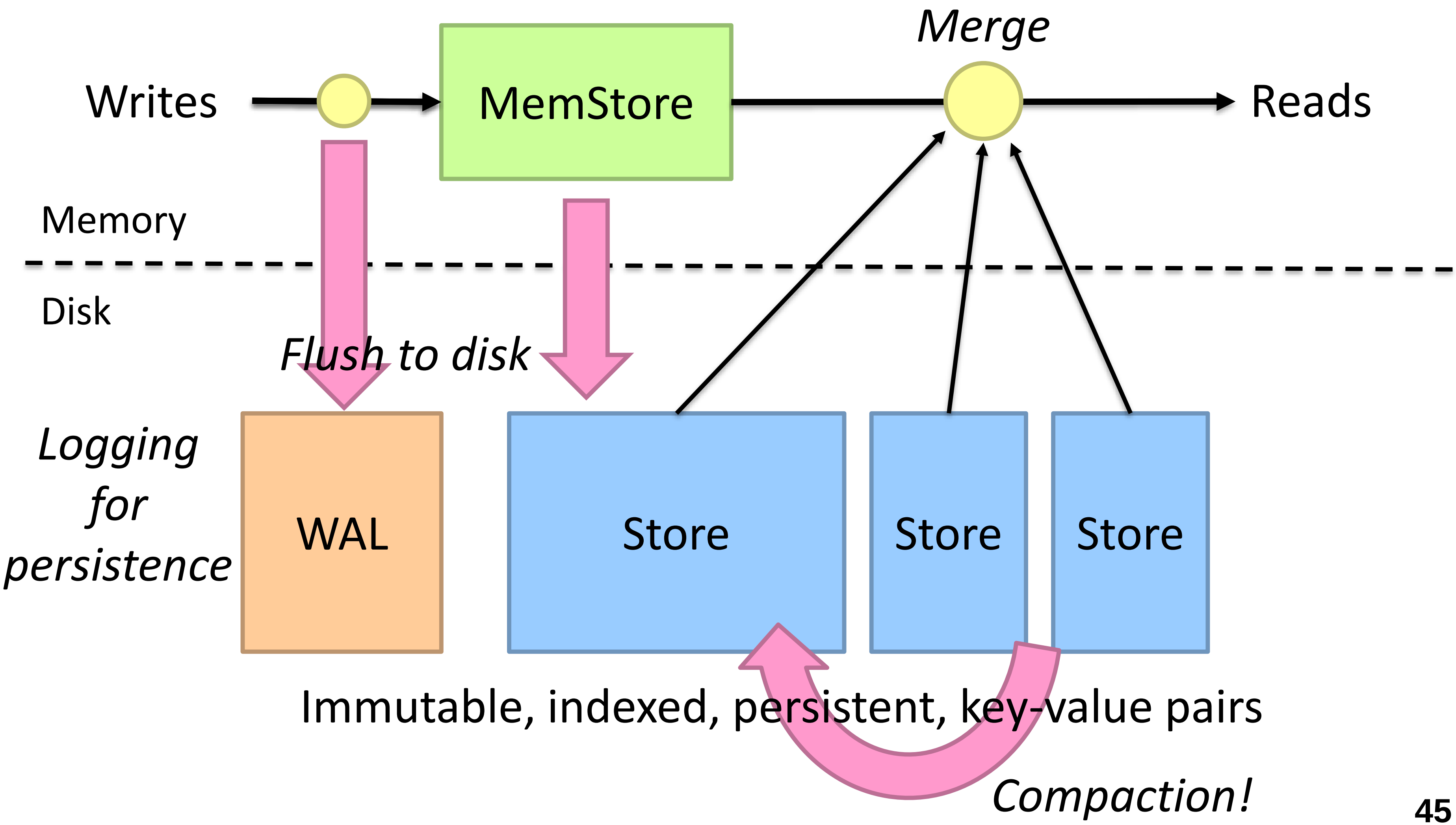


Immutable, indexed, persistent, key-value pairs

One final component…

Log Structured Merge Trees

The complete picture…



Immutable, indexed, persistent, key-value pairs

Log Structured Merge Trees

The complete picture…

Okay, now how do we build a distributed version?

HBase Bigtable building blocks

HDFS
GFS

HFile SSTable

Tablet Region

Regions Server
Tablet Server

Chubby
Zookeeper

SSTable HFile

Persistent, ordered immutable map from keys to values

Stored in GFS: replication "for free"

Supported operations:

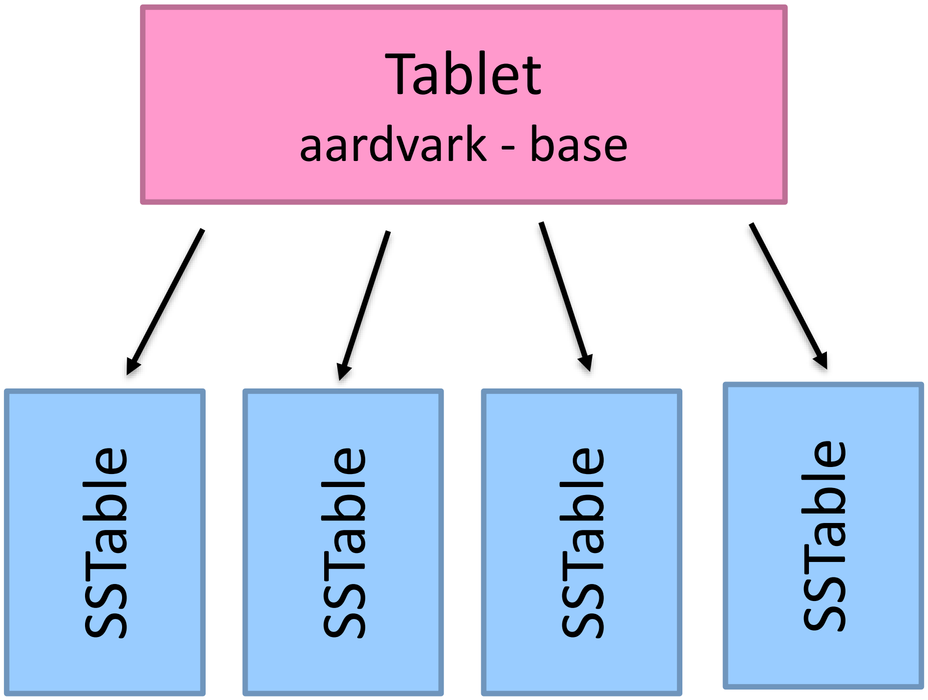
Look up value associated with key

Iterate key/value pairs within a key range

Region Tablet

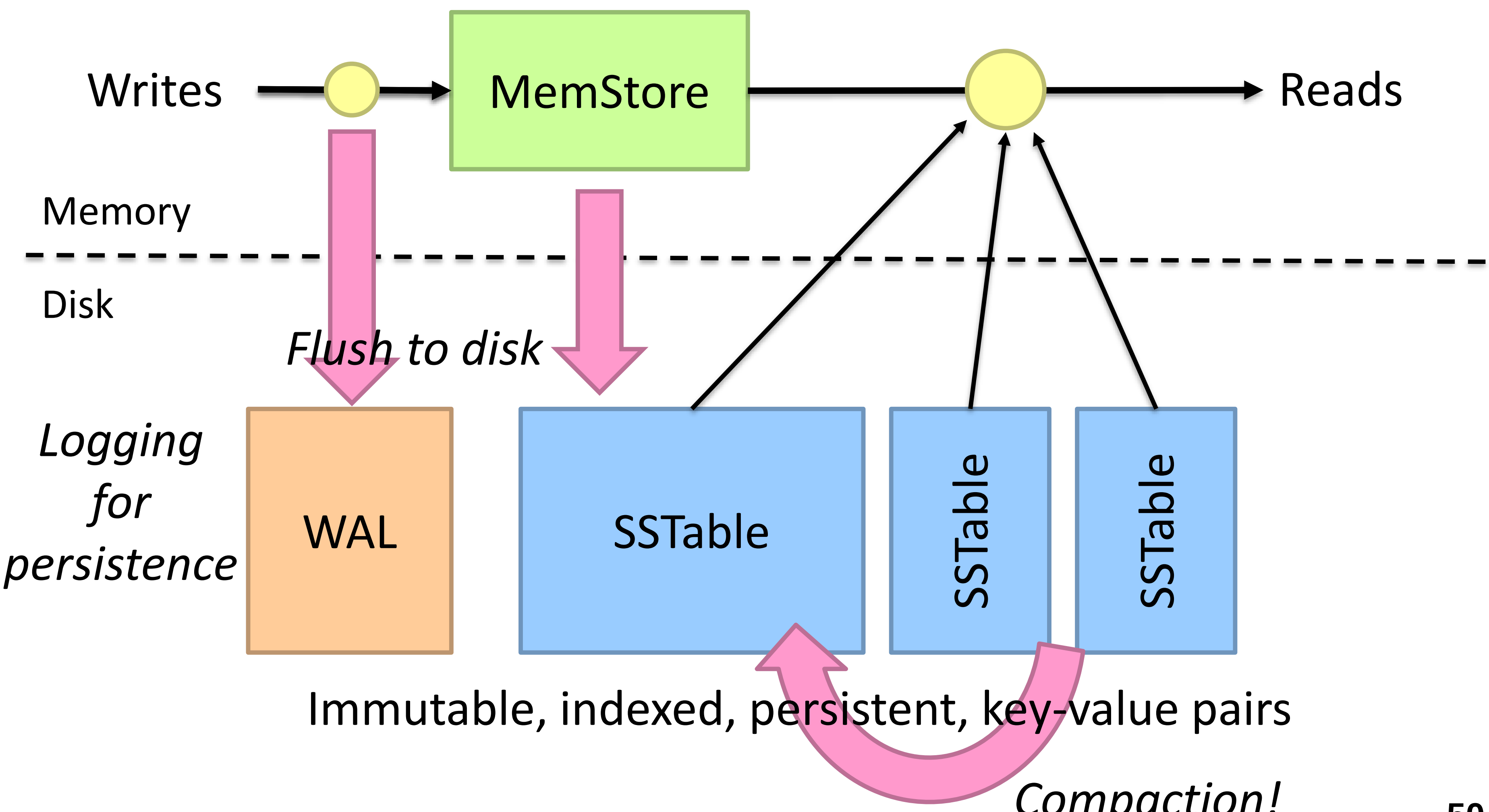
Dynamically partitioned range of rows

Comprised of multiple SSTables



Region Server

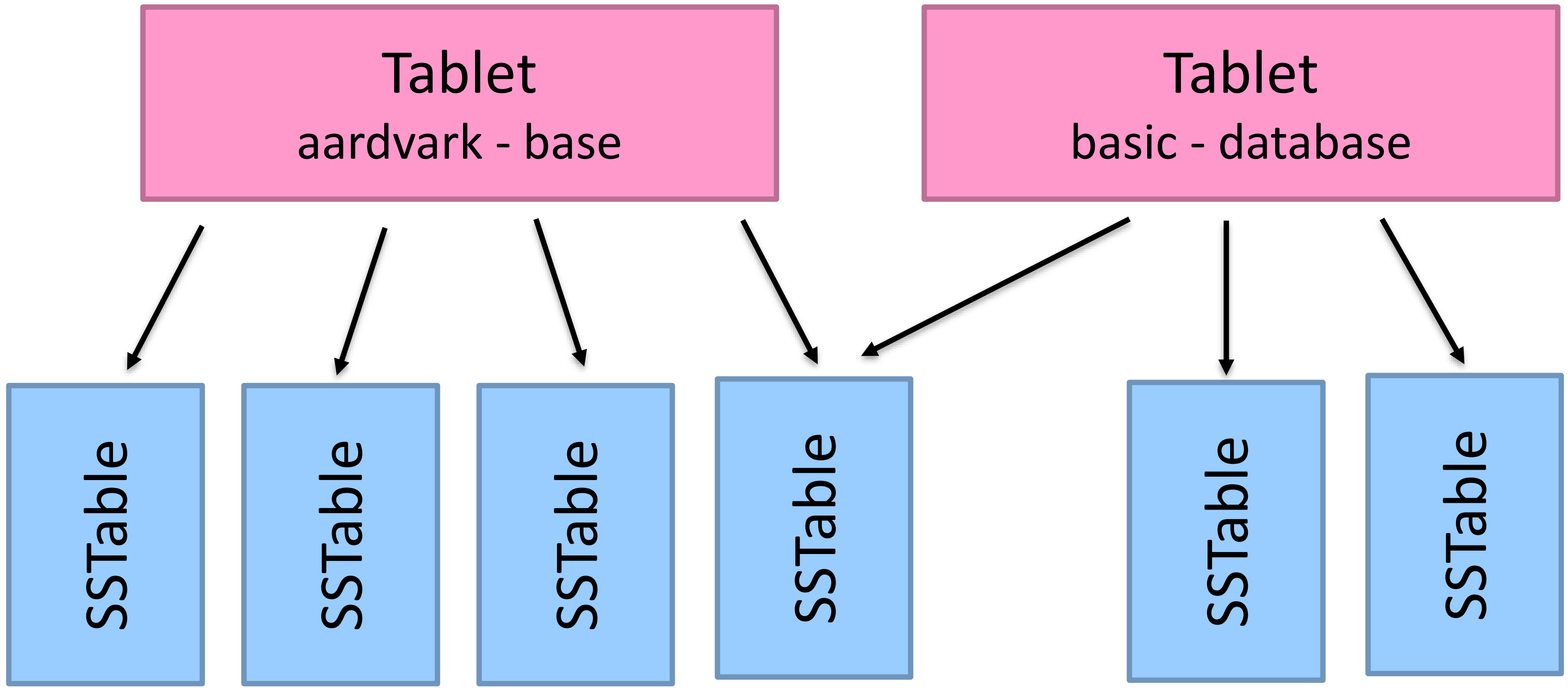
Tablet Server



Table

Comprised of multiple tablets

SSTables can be shared between tablets



Region

Region Server
Tablet to Tablet Server Assignment

Each tablet is assigned to one tablet server at a time

Exclusively handles read and write requests to that tablet

What happens when a tablet grow too big?

What happens when a tablet server fails?

We need a lock service!

HBase Bigtable building blocks

HDFS
GFS

HFile SSTable

Tablet Region

Regions Server
Tablet Server

Chubby
Zookeeper

Architecture

Client library

Bigtable master HMaster

Tablet servers

Regions Servers

Bigtable Master

Roles and responsibilities:

Assigns tablets to tablet servers

Detects addition and removal of tablet servers

Balances tablet server load

Handles garbage collection

Handles schema changes

Tablet structure changes:

Table creation/deletion (master initiated)

Tablet merging (master initiated)

Tablet splitting (tablet server initiated)

Compactions

Minor compaction

Converts the memtable into an SSTable

Reduces memory usage and log traffic on restart

Merging compaction

Reads a few SSTables and the memtable, and writes out a new SSTable

Reduces number of SSTables

Major compaction

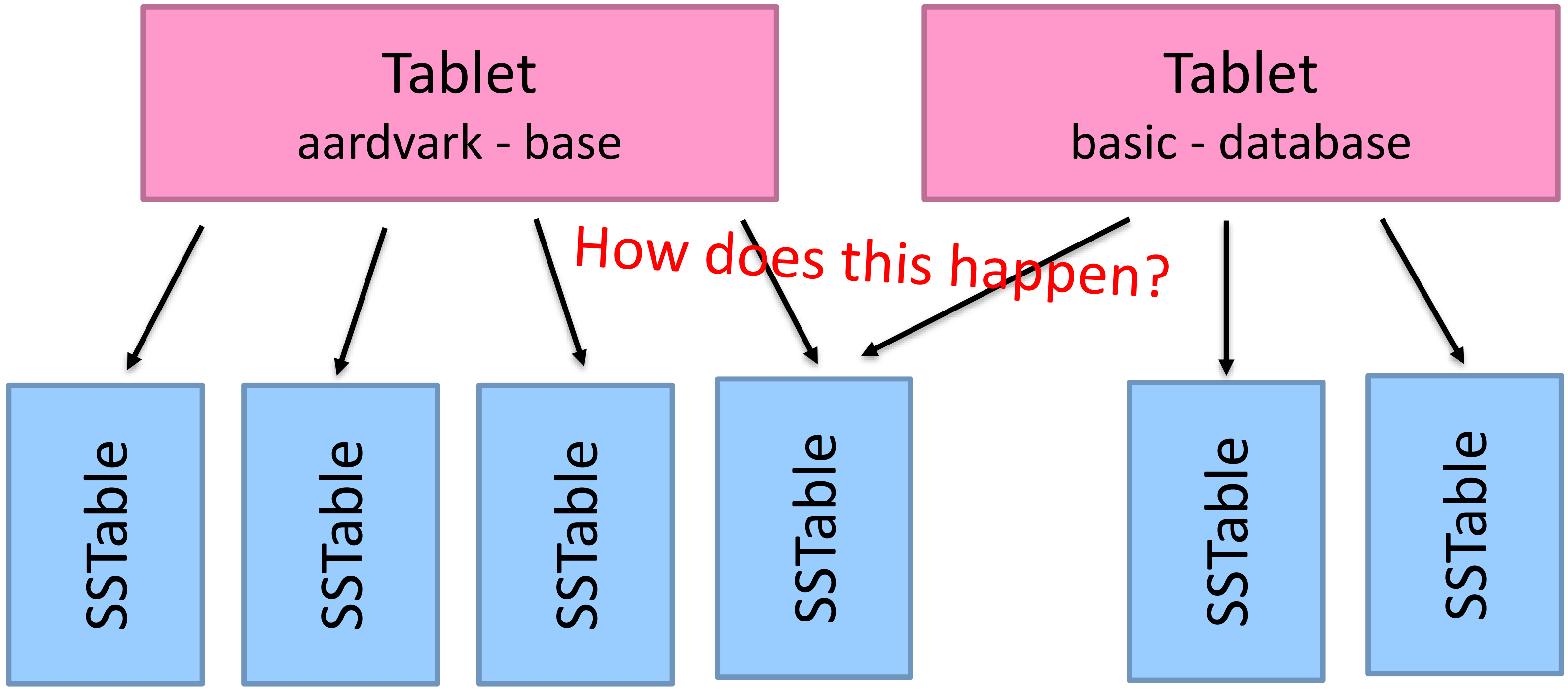
Merging compaction that results in only one SSTable

No deletion records, only live data

Table

Comprised of multiple tables

SSTables can be shared between tablets



Three Core Ideas

Keeping track of the partitions?

Partitioning (sharding)

To increase scalability and to decrease latency

Consistency?

Replication

To increase robustness (availability) and to increase throughput

Caching

To reduce latency

HBase

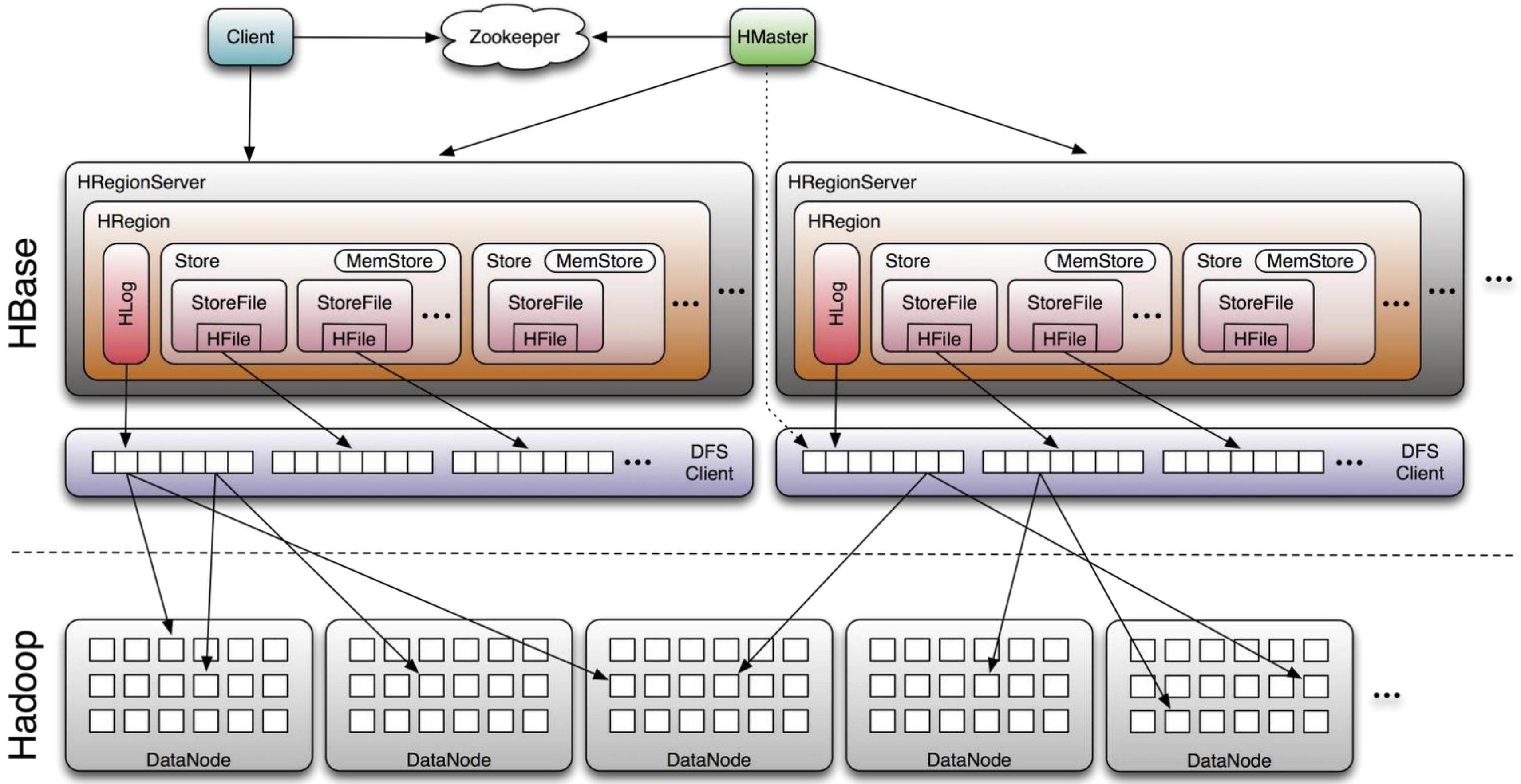


Image Source: http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html