POLITECNICO
MILANO 1863

SYSTEMS AND METHODS FOR BIG AND UNSTRUCTURED DATA

Columnar Databases – Cassandra

Marco Brambilla

marco.brambilla@polimi.it

@marcobrambi

Overview – NoSQL Family

Data stored in 4 types:

- Document
- Graph
- Key-value
- Wide-column

Document Database	Graph Databases
Couchbase, MarkLogic, mongoDB	Neo4j, InfiniteGraph – The Distributed Graph Database
Key-value Databases	**Wide Column Stores**
redis, amazon DynamoDB, AEROSPIKE, riak	accumulo, HYPERTABLE INC, Cassandra, APACHE HBASE, Amazon SimpleDB

Big Column Stores

Issues with today’s workloads

- Data large and unstructured
- Lots of random reads and writes
- Foreign keys rarely needed
- Actual Needs
  - Incremental Scalability
  - Speed
  - No Single point of failure
  - Low cost (TCO) and admin
  - Scale out, not up

CAP Theorem

- at most 2 out of the 3 guarantees
  1. Consistency: all nodes have same data at any time
  2. Availability: the system allows operations all the time
  3. Partition-tolerance: the system continues to work in spite of network partitions

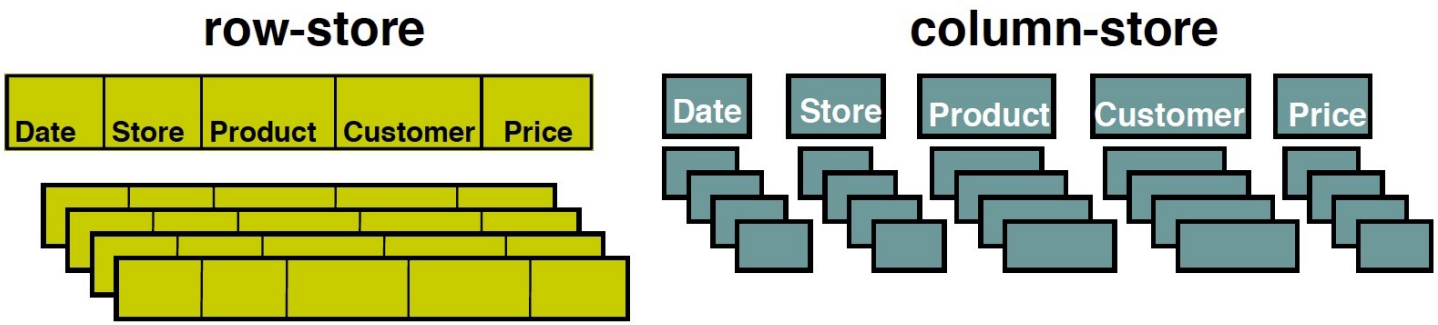
- Cassandra
  - Eventual (weak) consistency, Availability, Partition-tolerance

Introduction

- The Column Oriented Database stores data in columns .

- It is mainly used in OLAP(online Analytical Processing), Data Mining operations.

Column storage



+ easy to add/modify a record

- might read in unnecessary data

+ only need to read in relevant data

- tuple writes require multiple accesses

=> suitable for read-mostly, read-intensive, large data repositories

Pros

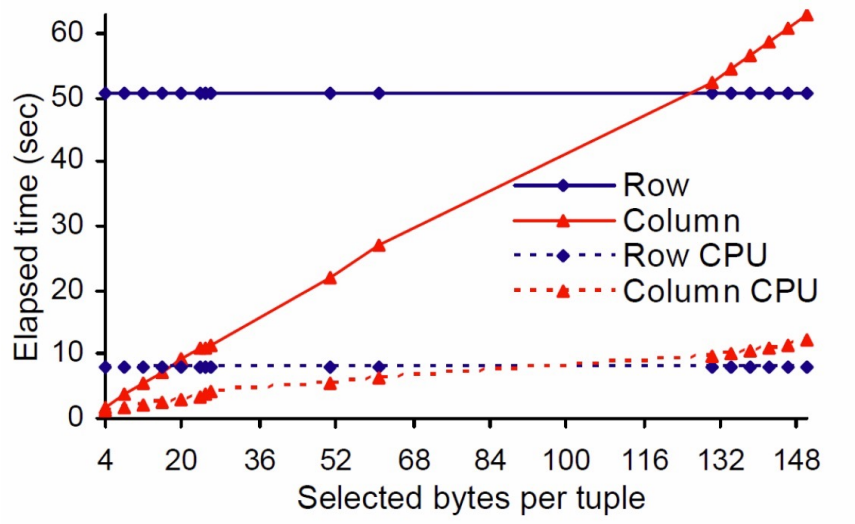
- Data compression
- Improved Bandwidth Utilization
- Improved Code Pipelining
- Improved cache locality

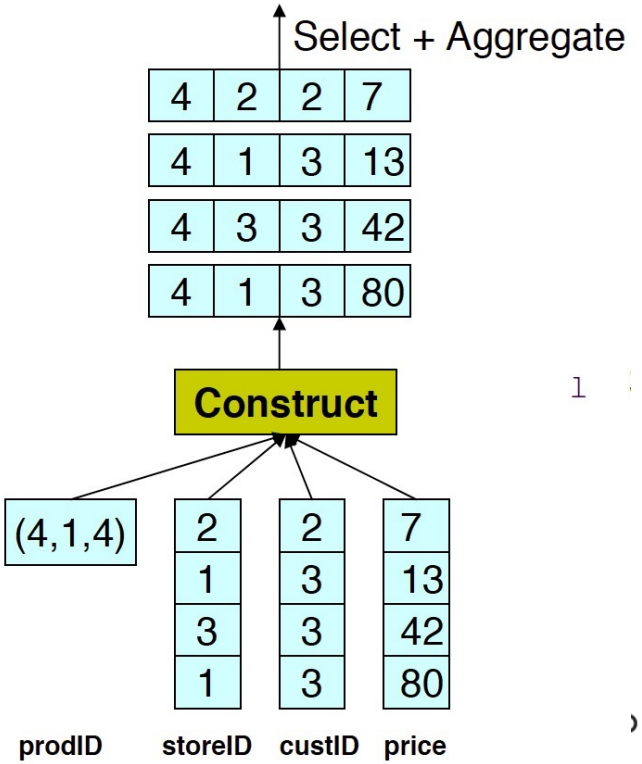
Cons

- Increased Disk Seek Time
- Increased cost of Inserts
- Increases tuple reconstruction costs

Tuple Reconstruction

- Large prefetch hides disk seeks in columns



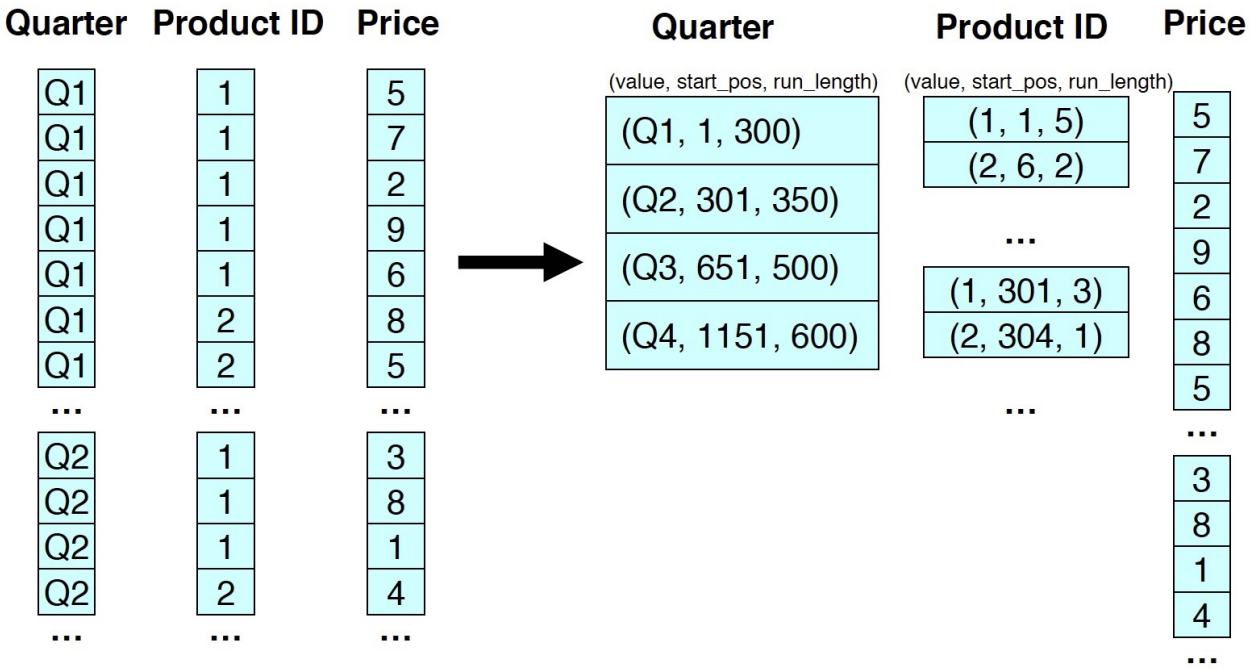


Compression

- Trades I/O for CPU
- Increased column-store opportunities:
- Higher data value locality in column stores
- Techniques such as run length encoding far more useful
- Can use extra space to store multiple copies of data in different sort orders

Compression: example

- Run-Length Encoding



List of Databases

- Cassandra
- Vertica
- SybaseIQ
- C-Store
- BigTable
- MonetDB
- LucidDB

Cassandra

Cassandra

- Originally designed at Facebook
- Open-sourced and now within Apache foundation
- Some of its users:

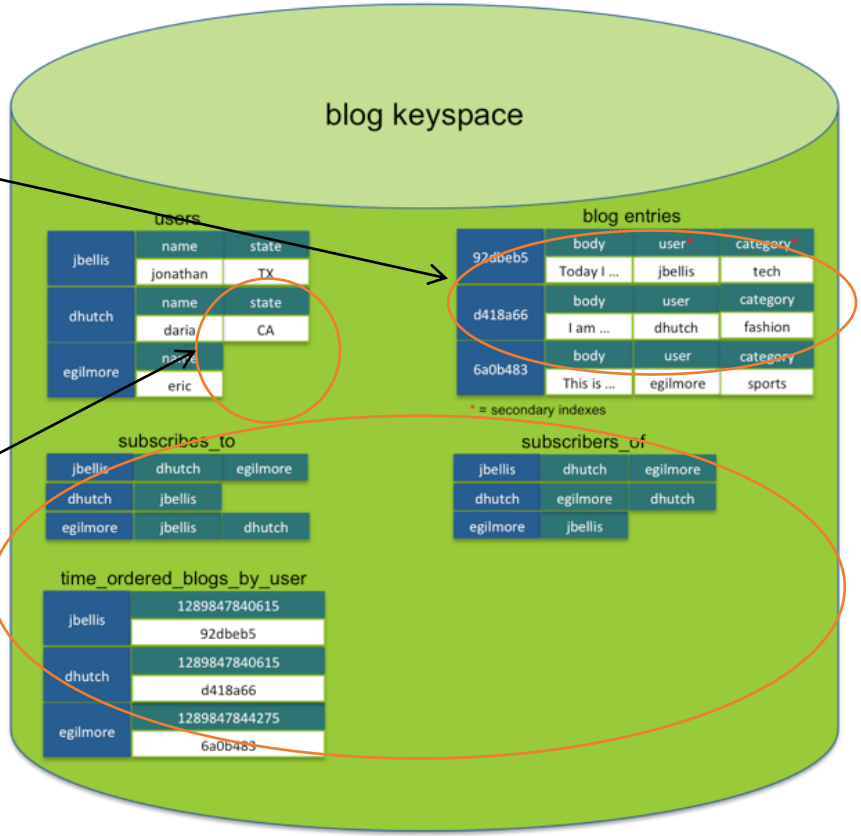
innovation at scale

- google bigtable (2006)
  - consistency model: strong
  - data model: sparse map
  - clones: hbase, hypertable
- amazon dynamo (2007)
  - O(1) dht
  - consistency model: client tune-able
  - clones: riak, voldemort

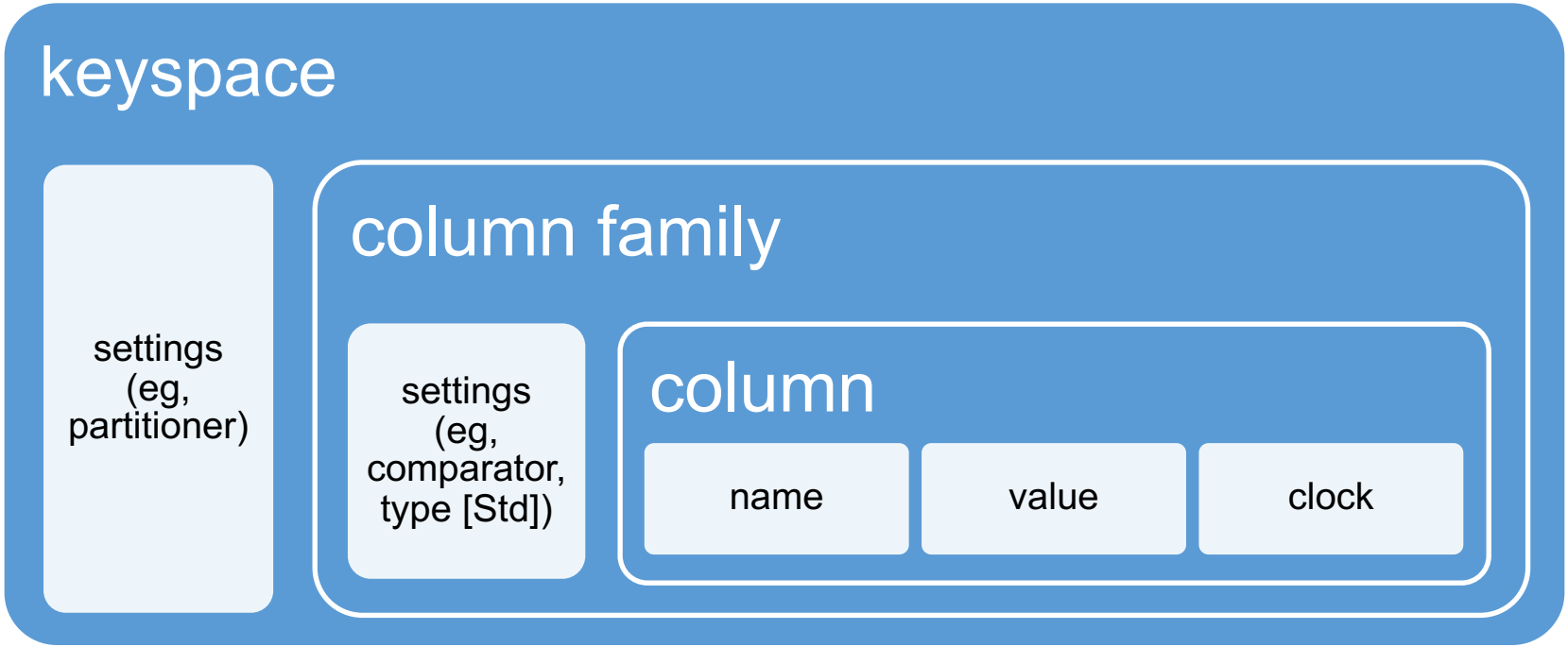
cassandra ~= bigtable + dynamo

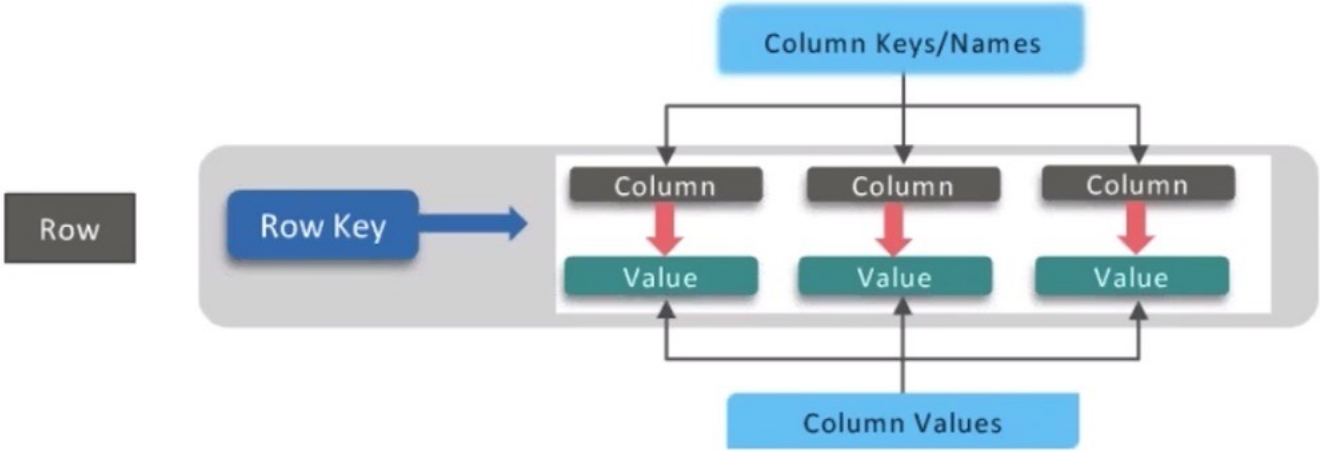
Cassandra Data Model

- Column Families:
  - Like SQL tables
  - but may be unstructured (client-specified)
  - Can have index tables
- Hence “column-oriented databases”/ “NoSQL”
  - No schemas
  - Some columns missing from some entries
  - “Not Only SQL”
  - Supports get(key) and put(key, value) operations
  - Often write-heavy workloads



Data model


keyspace
settings (eg, partitioner)
column family
settings (eg, comparator, type [Std])
column
name
value
clock


Column Keys/Names
Row
Row Key
Column
Column
Column
Value
Value
Value
Column Values

keyspace

- ~= database
- typically one per application
- some settings are configurable only per keyspace

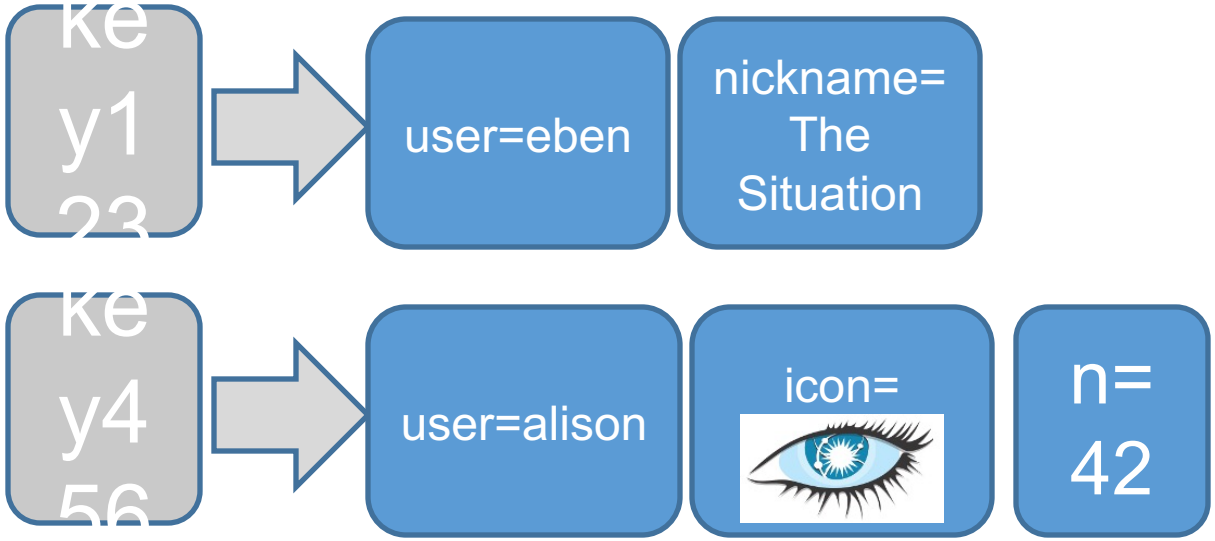
column family

- group records of *similar* kind
- not *same* kind, because CFs are sparse tables
- ex:
  - User
  - Address
  - Tweet
  - PointOfInterest
  - HotelRoom

think of cassandra as
row-oriented

- each row is uniquely identifiable by key
- rows group columns and super columns

column family



json-like notation

User {

123 : { email: alison@foo.com,

icon: 

456 : { email: eben@bar.com,

location: The Danger Zone}

}

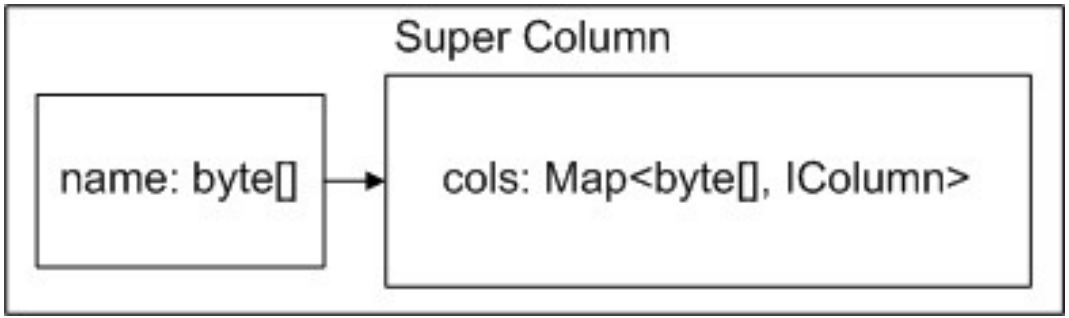
a column has 3 parts

1. name
   - byte[]
   - determines sort order
   - used in queries
   - indexed
2. value
   - byte[]
   - *you don't query on column values*
3. timestamp
   - long (clock)
   - last write wins conflict resolution

column comparators

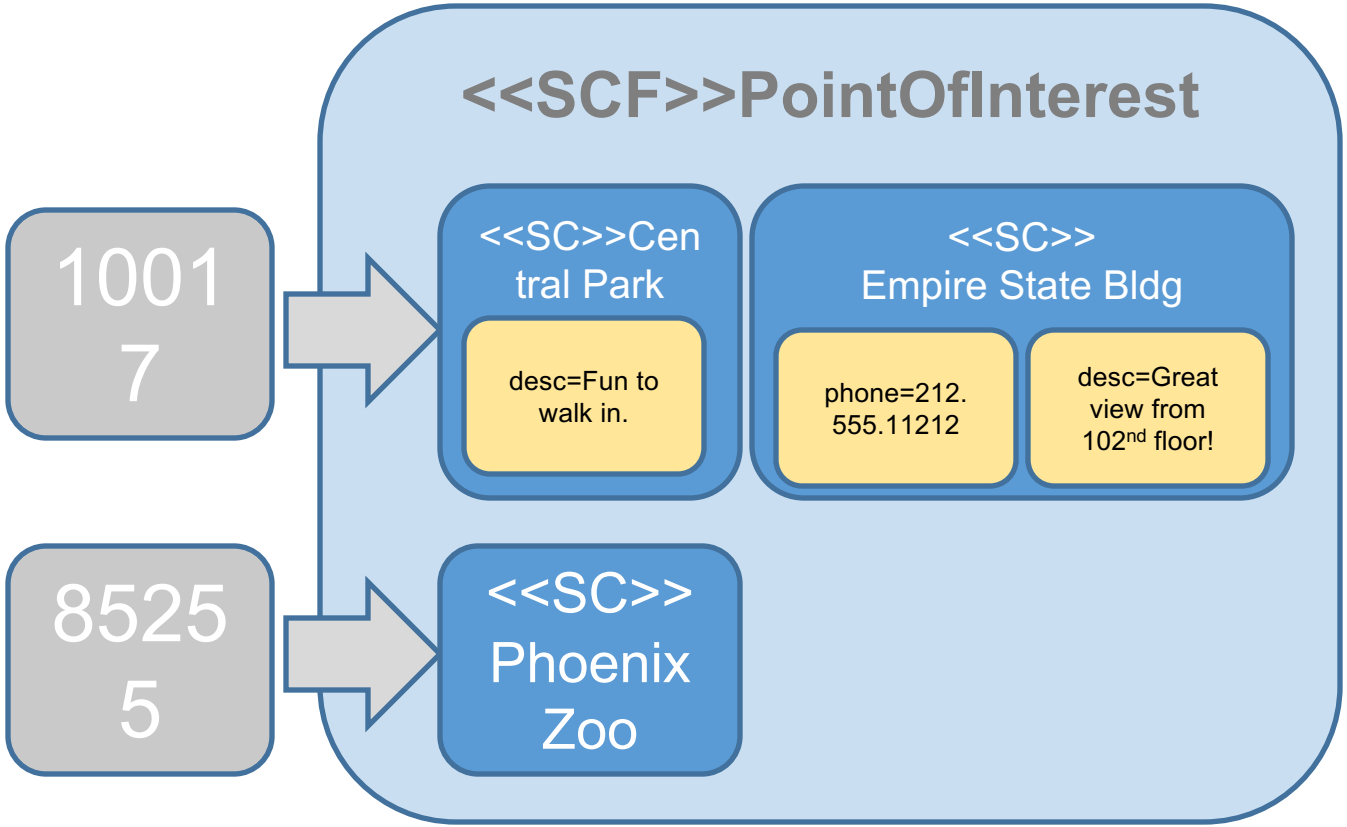
- byte
- utf8
- long
- timeuuid
- lexicaluuid
- <pluggable>
  - ex: lat/long

super column

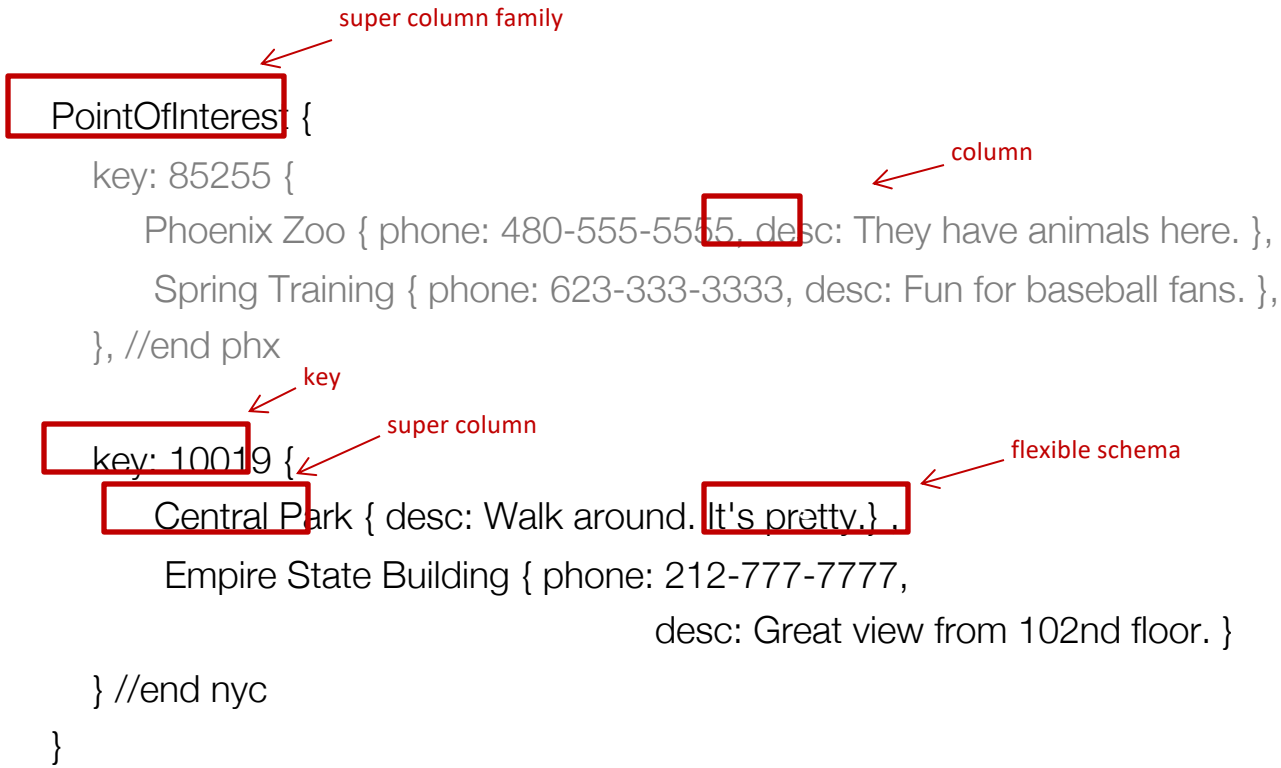


super columns group columns under a common name

super column family



super column family



about super column families

- sub-column names in a SCF are *not* indexed
  - top level columns (SCF Name) are *always* indexed
- often used for denormalizing data from standard CFs

slice predicate

- data structure describing columns to return
  - SliceRange
    - start column name
    - finish column name (can be empty to stop on count)
    - reverse
    - count (like LIMIT)

read api

- get() : Column
  - get the Col or SC at given ColPath

```
COSC cosc = client.get(key, path, CL);
```

- get_slice() : List<ColumnOrSuperColumn>
  - get Cols in *one* row, specified by SlicePredicate:

```
List<ColumnOrSuperColumn> results =
  client.get_slice(key, parent, predicate, CL);
```

- multiget_slice() : Map<key, List<CoSC>>
  - get slices for *list of keys*, based on SlicePredicate

```
Map<byte[],List<ColumnOrSuperColumn>> results =
    client.multiget_slice(rowKeys, parent, predicate, CL);
```

- get_range_slices() : List<KeySlice>
  - returns multiple Cols according to a *range*
  - range is startkey, endkey, starttoken, endtoken:

```
List<KeySlice> slices = client.get_range_slices(
    parent, predicate, keyRange, CL);
```

write api

```
client.insert(userKeyBytes, parent,
        new Column(“band".getBytes(UTF8),
        “Funkadelic".getBytes(), clock), CL);
```

batch_mutate

- void batch_mutate(
  map<byte[], map<String, List<Mutation>>> , CL)

remove

- void remove(byte[],
  ColumnPath column_path, Clock, CL)

what about… SQL?

SELECT
WHERE
ORDER BY
JOIN ON
GROUP

rdbms: domain-based model

what answers do I have?

cassandra: query-based model

what *questions* do I have?

Start from queries, then design the data model

SELECT WHERE

cassandra is an index factory

<<cf>>USER
Key: UserID
Cols: username, email, birth date, city, state

How to support this query?

```
SELECT * FROM User WHERE city = 'Scottsdale'
```

Create a new CF called UserCity:

<<cf>>USERCITY
Key: city
Cols: IDs of the users in that city.
Also uses the Valueless Column pattern

SELECT WHERE pt 2

- Use an aggregate key
  `state:city: { user1, user2}`

- Get rows between **`AZ:`** & **`AZ;`**
  for all Arizona users

- Get rows between **`AZ:Scottsdale`** & **`AZ:Scottsdale1`**
  for all Scottsdale users

ORDER BY

Columns

are sorted according to

`CompareWith` or `CompareSubcolumnsWith`

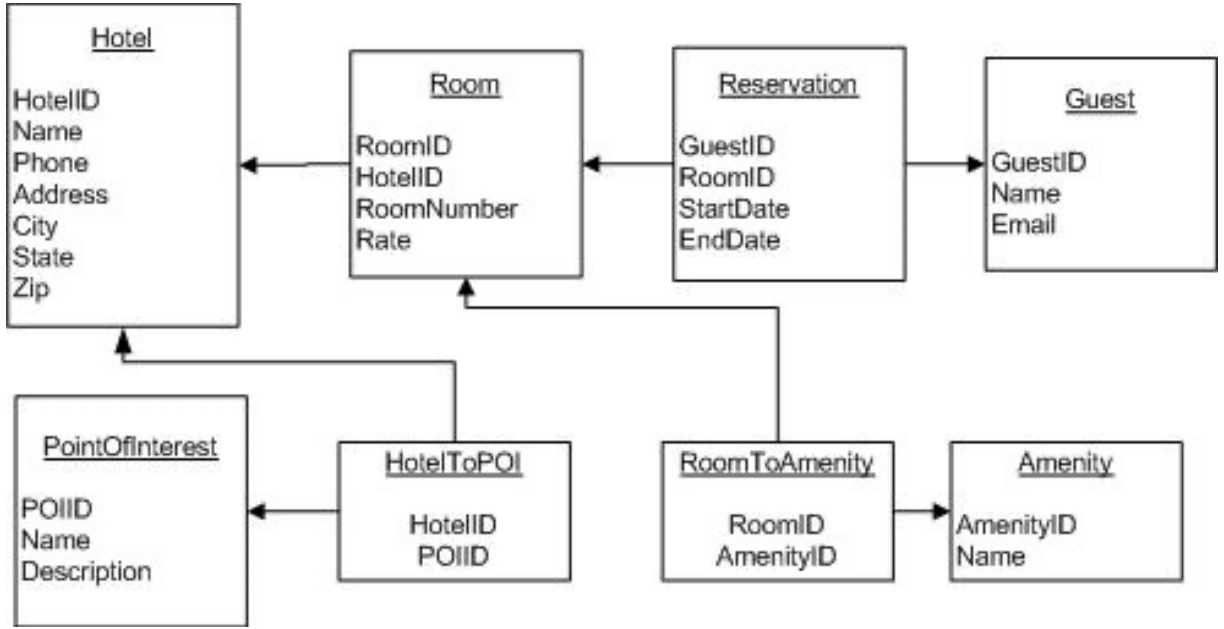
Rows

are *placed* according to their Partitioner:

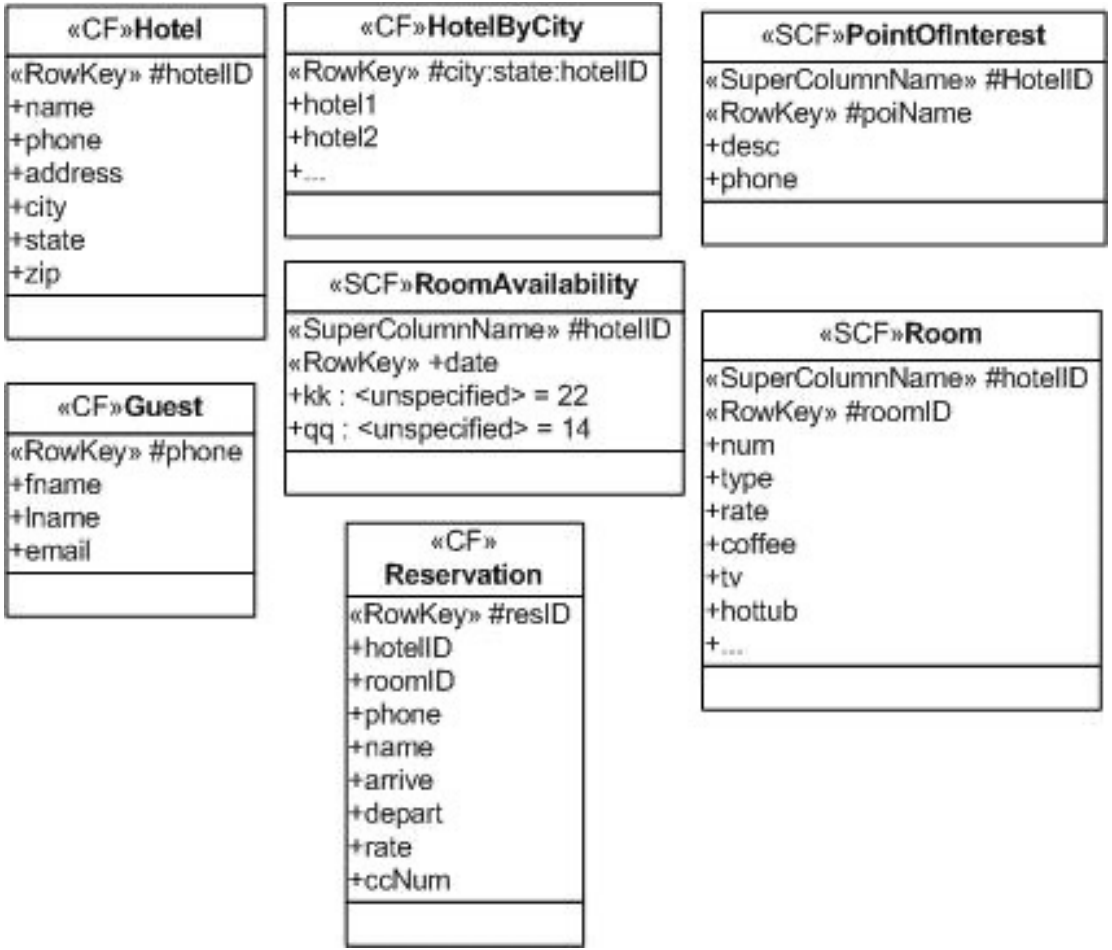
- Random: MD5 of key
- Order-Preserving: actual key

are *sorted* by key, regardless of partitioner

rdbms


Hotel
HotelID
Name
Phone
Address
City
State
Zip
Room
RoomID
HotelID
RoomNumber
Rate
Reservation
GuestID
RoomID
StartDate
EndDate
Guest
GuestID
Name
Email
PointOfInterest
POIID
Name
Description
HotelToPOI
HotelID
POIID
RoomToAmenity
RoomID
AmenityID
Amenity
AmenityID
Name

cassandra


«CF»Hotel
«RowKey» #hotelID
+name
+phone
+address
+city
+state
+zip
«CF»HotelByCity
«RowKey» #city:state:hotelID
+hotel1
+hotel2
+...
«SCF»PointOfInterest
«SuperColumnName» #HotelID
«RowKey» #poiName
+desc
+phone
«SCF»RoomAvailability
«SuperColumnName» #hotelID
«RowKey» +date
+kk : <unspecified> = 22
+qq : <unspecified> = 14
«CF»Guest
«RowKey» #phone
+fname
+lname
+email
«SCF»Room
«SuperColumnName» #hotelID
«RowKey» #roomID
+num
+type
+rate
+coffee
+tv
+hottub
+...
«CF»
Reservation
«RowKey» #resID
+hotelID
+roomID
+phone
+name
+arrive
+depart
+rate
+ccNum

Properties wrt RDBMS (e.g., Oracle)

Property	Cassandra	RDBMS
Core Architecture	Masterless (no single point of failure)	Master-slave (single points of failure)
High Availability	Always-on continuous availability	General replication with master-slave
Data Model	Dynamic; structured and unstructured data	Legacy RDBMS; Structured data
Scalability Model	Big data/Linear scale performance	Oracle RAC or Exadata
Multi-Data Center Support	Multi-directional, multi-cloud availability	Nothing specific
Enterprise Search	Integrated search on Cassandra data.	Handled via Oracle search
In-Memory Database Option	Built-in in-memory option	Columnar in-memory option

Properties wrt RDBMS (e.g., Oracle)

Property	Cassandra	RDBMS
Joining	Doesn't support joining	Supports joining
Referential Integrity	Cassandra has no concept of referential integrity across tables. No cascading deletes.	Supports foreign keys in a table to reference the primary key of a another table. Supports cascading delete.
Normalization	Tables contain duplicate denormalize data.	Tables are normalized to avoid redundancy.

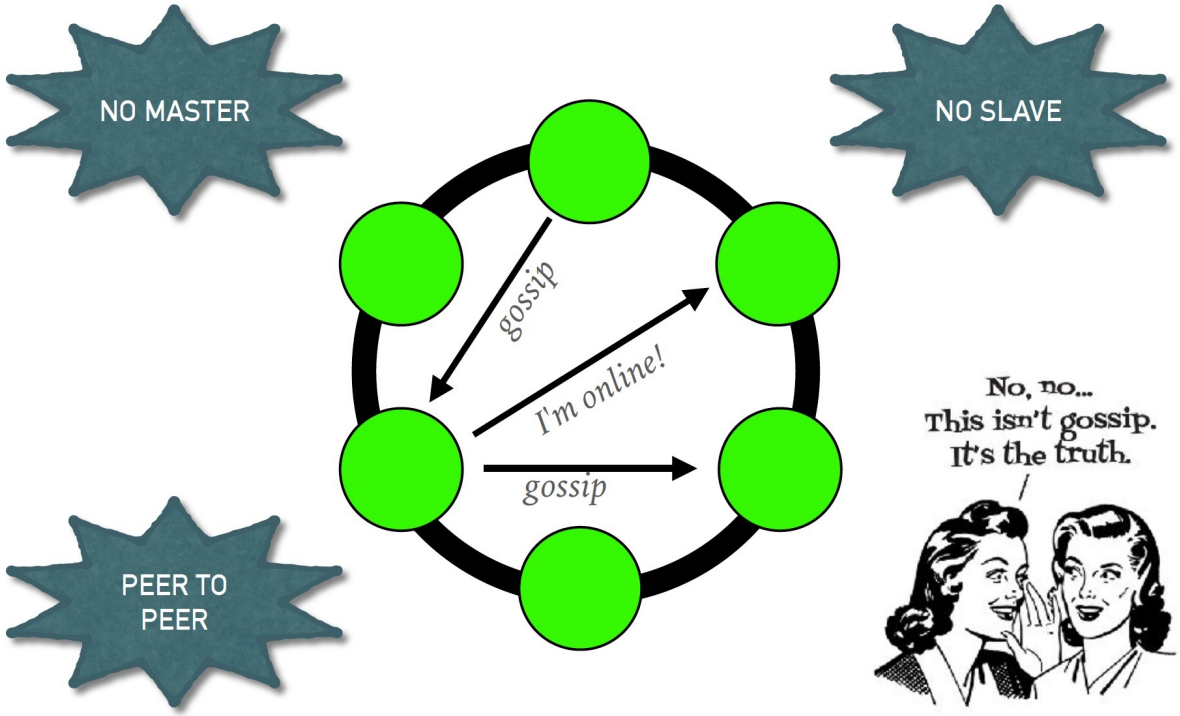
cassandra properties

- *tuneably* consistent
- very fast writes
- highly available
- fault tolerant
- linear, elastic scalability
- decentralized/symmetric
- ~12 client languages
  - Thrift RPC API
- ~automatic provisioning of new nodes
- 0(1) dht
- big data

Let’s go Inside: Key -> Server Mapping

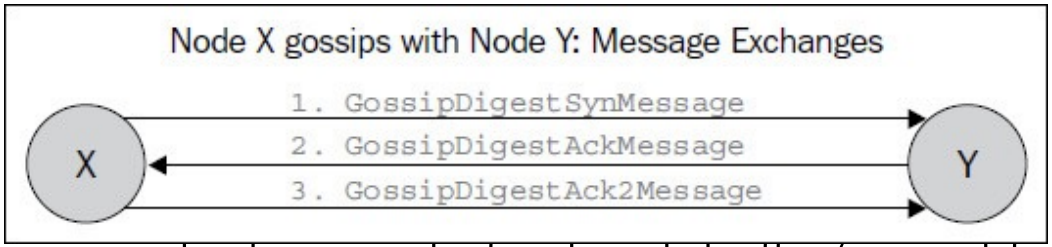
- How do you decide which server(s) a key-value resides on?

Do you like gossips?



Gossip protocol

- Each node picks its discussants (up to 3)
- Having three messages for each round of gossip adds a degree of *anti-entropy*.
- This process allows obtaining "convergence" of data shared between the two interacting nodes much faster.



- Always a constant amount of network traffic (except for gossip storms)

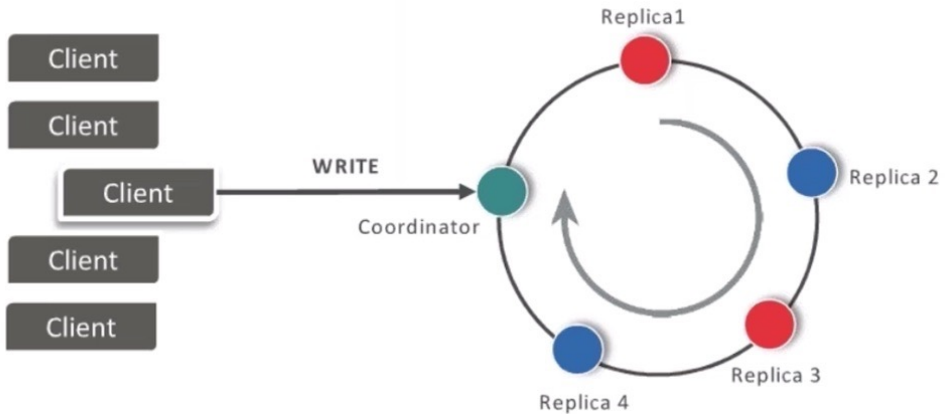
Replica placement strategies

- Simple Strategy
  - Single datacenter
  - Clockwise placement to the next node(s)
- Network Topology

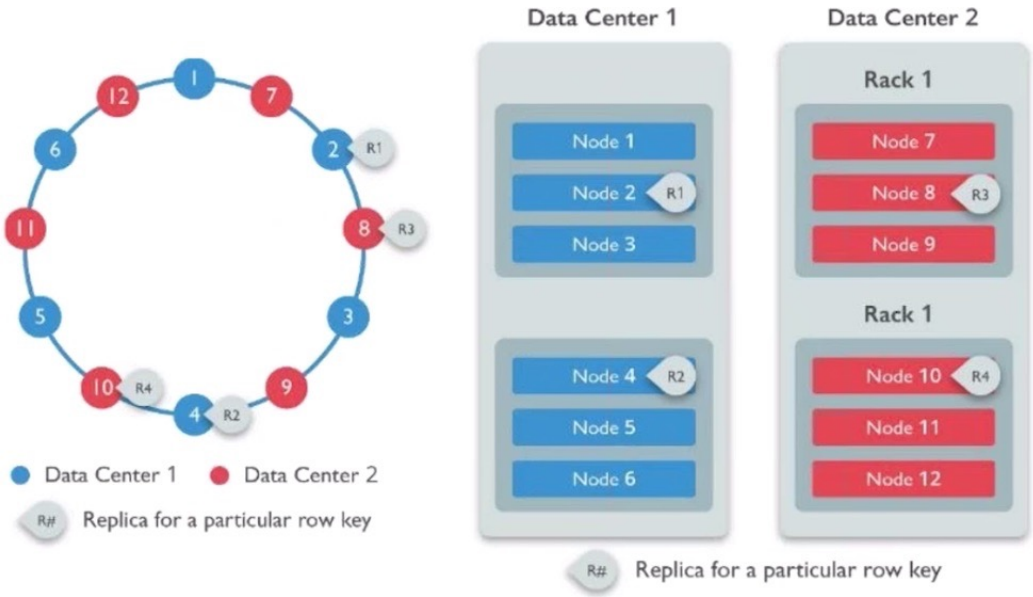
Strategy

  - Multiple datacenters
  - Supporting local queries

Simple Strategy: Ring



Network Topology Strategy



Writes

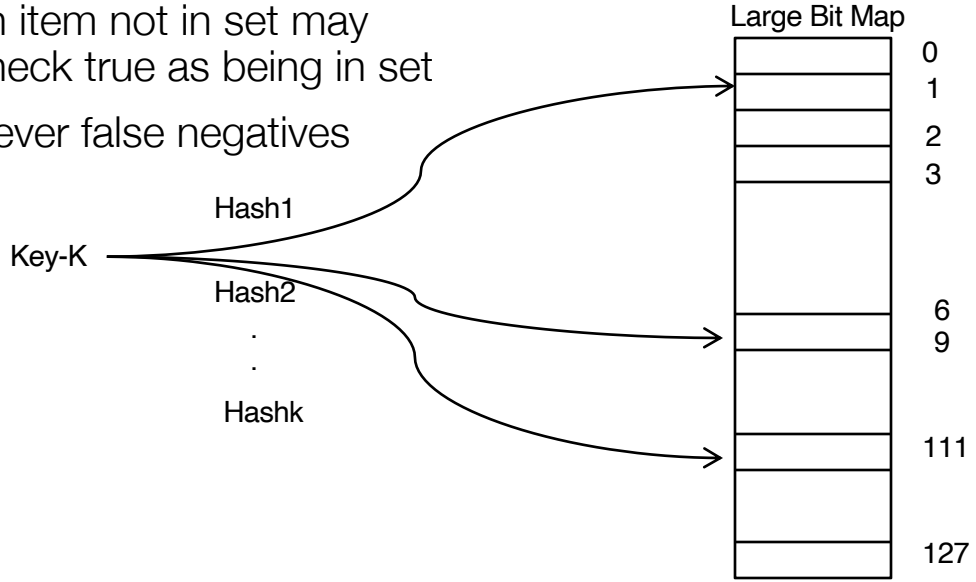
- Need to be lock-free and fast (no reads or disk seeks)
- Client sends write to one front-end node in Cassandra cluster (Coordinator)
- Which (via Partitioning function) sends it to all replica nodes responsible for key
  - Always writable: Hinted Handoff
    - If any replica is down, the coordinator writes to all other replicas, and keeps the write until down replica comes back up.
    - When all replicas are down, the Coordinator (front end) buffers writes (for up to an hour).
  - Provides Atomicity for a given key (i.e., within ColumnFamily)
- One ring per datacenter
  - Coordinator can also send write to one replica per remote datacenter

Writes at a replica node

- On receiving a write
- 1. log it in disk commit log
- 2. Make changes to appropriate memtables
  - In-memory representation of multiple key-value pairs
- Later, when memtable is full or old, flush to disk
  - Data File: An SSTable (Sorted String Table) – list of key value pairs, sorted by key
  - Index file: An SSTable – (key, position in data sstable) pairs
    - And a Bloom filter
- Compaction: Data udpates accumulate over time and sstables and logs need to be compacted
  - Merge key updates, etc.
- Reads need to touch log and multiple SSTables
  - May be slower than writes

Bloom Filter

- Compact way of representing a set of items
- Checking for existence in set is cheap
- Some probability of false positives:
  an item not in set may
  check true as being in set
- Never false negatives



On insert, set all hashed bits.

On check-if-present, return true if all hashed bits set.

- False positives

False positive rate low

- k=4 hash functions
- 100 items
- 3200 bits
- FP rate = 0.02%

Deletes and Reads

- Delete: don’t delete item right away
  - add a tombstone to the log
  - Compaction will remove tombstone and delete item

- Read: Similar to writes, except
  - Coordinator can contact closest replica (e.g., in same rack)
  - Coordinator also fetches from multiple replicas
    - check consistency in the background, initiating a read-repair if any two values are different
    - Makes read slower than writes (but still fast)
    - Read repair: uses gossip

Cassandra uses Quorums

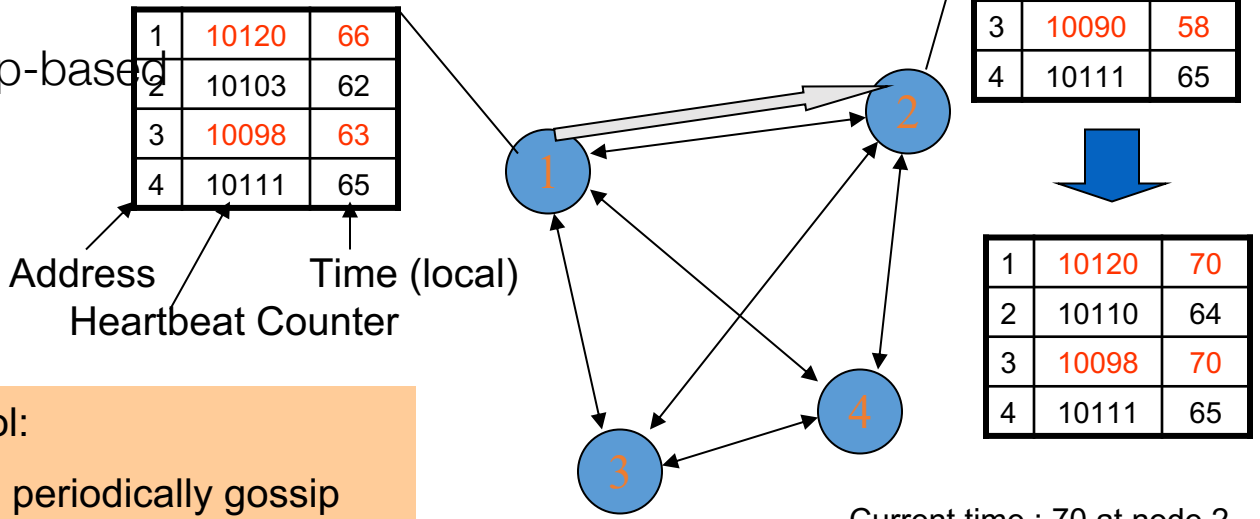
- Reads
  - Wait for R replicas (R specified by clients)
  - In background check for consistency of remaining N-R replicas, and initiate read repair if needed (N = total number of replicas for this key)
- Writes come in two flavors
  - Block until quorum is reached
  - Async: Write to any node
- Quorum Q = N/2 + 1
- R = read replica count, W = write replica count
- If W+R > N and W > N/2, you have consistency
- Allowed (W=1, R=N) or (W=N, R=1) or (W=Q, R=Q)

Cassandra uses Quorums

- In reality, a client can choose one of these levels for a read/write operation:
  - ANY: any node (may not be replica)
  - ONE: at least one replica
  - QUORUM: quorum across all replicas in all datacenters
  - LOCAL_QUORUM: in coordinator's DC
  - EACH_QUORUM: quorum in every DC
  - ALL: all replicas all DCs

Cluster Membership

- Gossip-based



Protocol:

•Nodes periodically gossip their membership list

•On receipt, the local membership list is updated

Current time : 70 at node 2

(asynchronous clocks)

Cluster Membership, contd.

- Suspicion mechanisms
- Accrual detector: FD outputs a value (PHI) representing suspicion
- Apps set an appropriate threshold
- PHI = 5 => 10-15 sec detection time
- PHI calculation for a member
  - Inter-arrival times for gossip messages
  - PHI(t) = - log(CDF or Probability(t_now – t_last))/log 10
  - PHI basically determines the detection timeout, but is sensitive to actual inter-arrival time variations for gossiped heartbeats

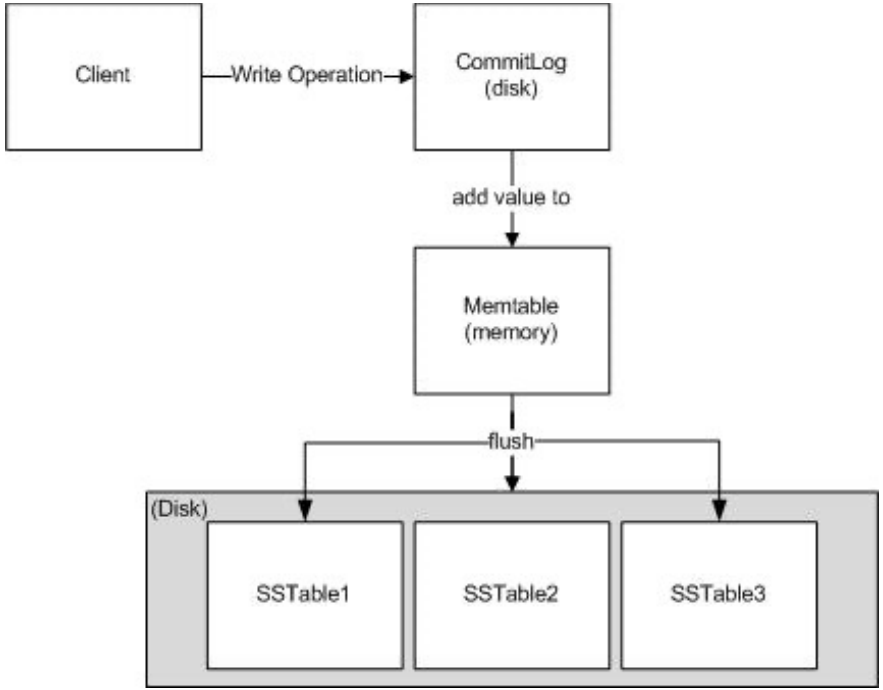
write consistency

Level	Description
ZERO	Good luck with that
ANY	1 replica (hints count)
ONE	1 replica. read repair in bkgnd
QUORUM (DCQ for RackAware)	(N /2) + 1
ALL	N = replication factor

read consistency

Level	Description
ZERO	Ummm…
ANY	Try ONE instead
ONE	1 replica
QUORUM (DCQ for RackAware)	Return most recent TS after (N /2) + 1 report
ALL	N = replication factor

write op


Client
Write Operation
CommitLog
(disk)
add value to
Memtable
(memory)
flush
(Disk)
SSTable1
SSTable2
SSTable3

Staged Event-Driven Architecture

- A general-purpose framework for high concurrency & load conditioning
- Decomposes applications into stages separated by queues
- Adopt a structured approach to event-driven concurrency

data replication

- configurable replication factor
- replica placement strategy
  - ~~rack unaware~~ → Simple Strategy
  - ~~rack aware~~ → Old Network Topology Strategy
  - ~~data center shard~~ → Network Topology Strategy

partitioner smack-down

Random Preserving

- system will use MD5(key) to distribute data across nodes
- even distribution of keys from one CF across ranges/nodes

Order Preserving

- key distribution determined by token
- lexicographical ordering
- required for range queries
  - scan over rows like cursor in index
- can specify the token for this node to use
- ‘scrabble’ distribution

is cassandra a good fit?

- you need really fast writes
- you need durability
- you have lots of data

  > GBs

  >= three servers
- your app is evolving
  - startup mode, fluid data structure
- loose domain data
  - “points of interest”
- your programmers can deal
  - documentation
  - complexity
  - consistency model
  - change
  - visibility tools
- your operations can deal
  - hardware considerations
  - can move data
  - JMX monitoring

Vs. SQL

- On > 50 GB data
- MySQL
  - Writes 300 ms avg
  - Reads 350 ms avg
- Cassandra
  - Writes 0.12 ms avg
  - Reads 15 ms avg

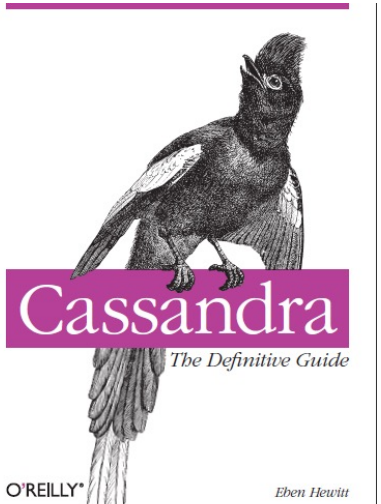
Cassandra Summary

- BASE properties
- Basically Available Soft-state Eventual Consistency

  - Cassandra
    - Prefers Availability over Consistency
  - HBase
    - Prefers (strong) Consistency over Availability

References

- https://medium.com/@michaeljpr/five-minute-guide-getting-started-with-cassandra-on-docker-4ef69c710d84



HBase

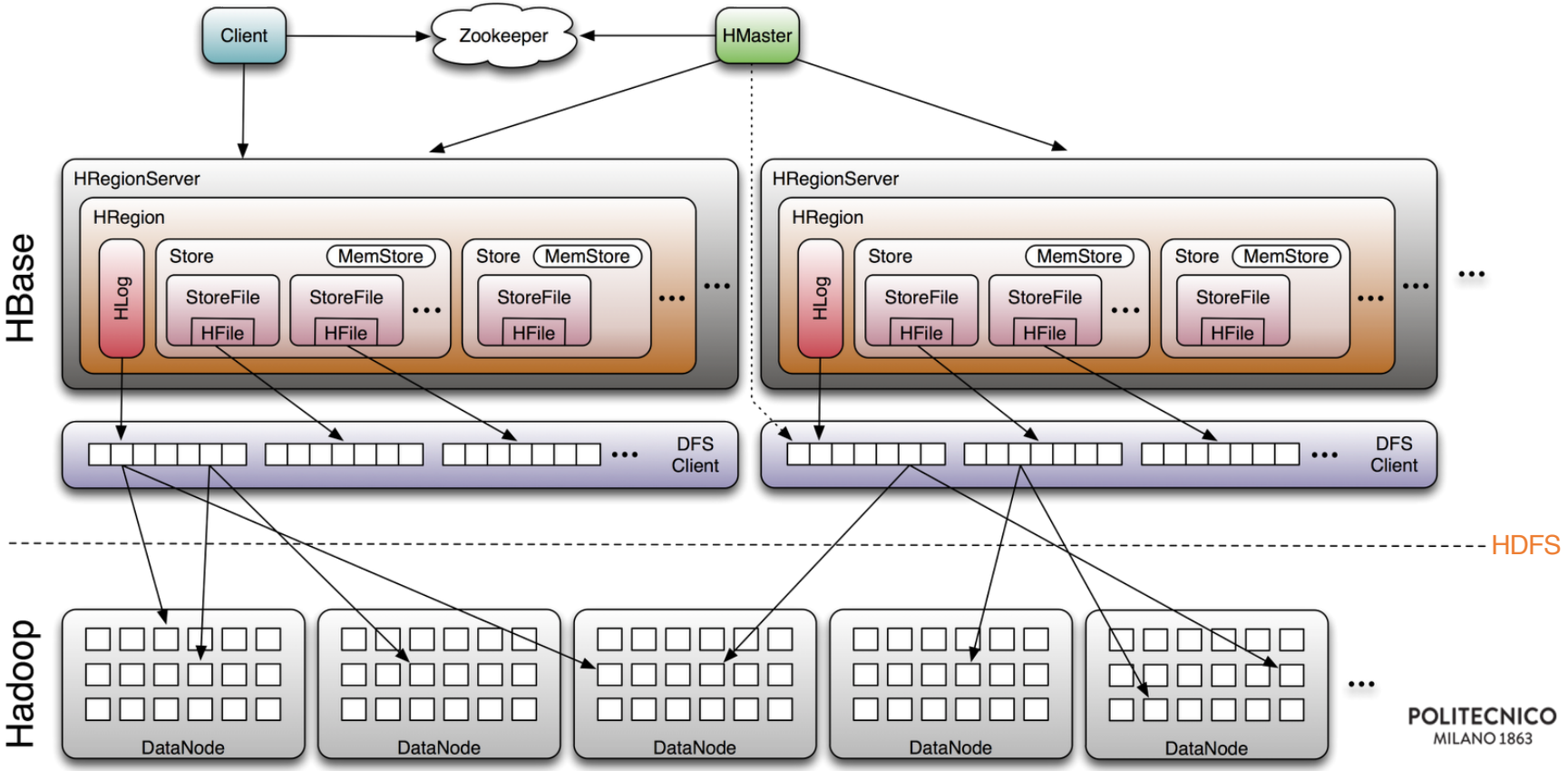
HBase

- Google's BigTable was first "blob-based" storage system
- Yahoo! Open-sourced it -> HBase
- Major Apache project today, part of Hadoop
- Facebook uses HBase internally
- API
  - Get/Put(row)
  - Scan(row range, filter) – range queries
  - MultiPut

Hbase API

- Supported operations
  - Get(row)
  - Put(row)
  - Scan(row range, filter) – range queries
  - MultiPut(rows)

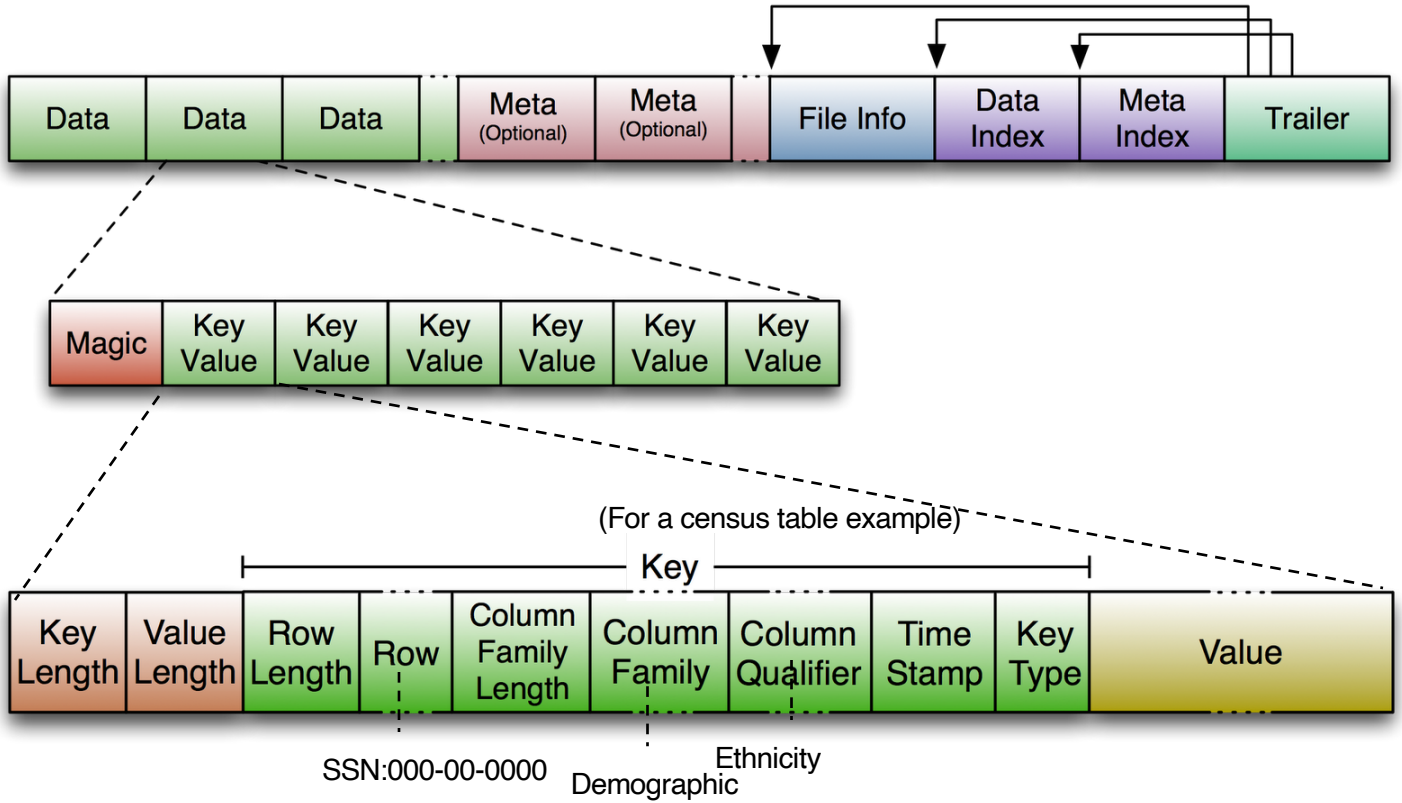
HBase Architecture



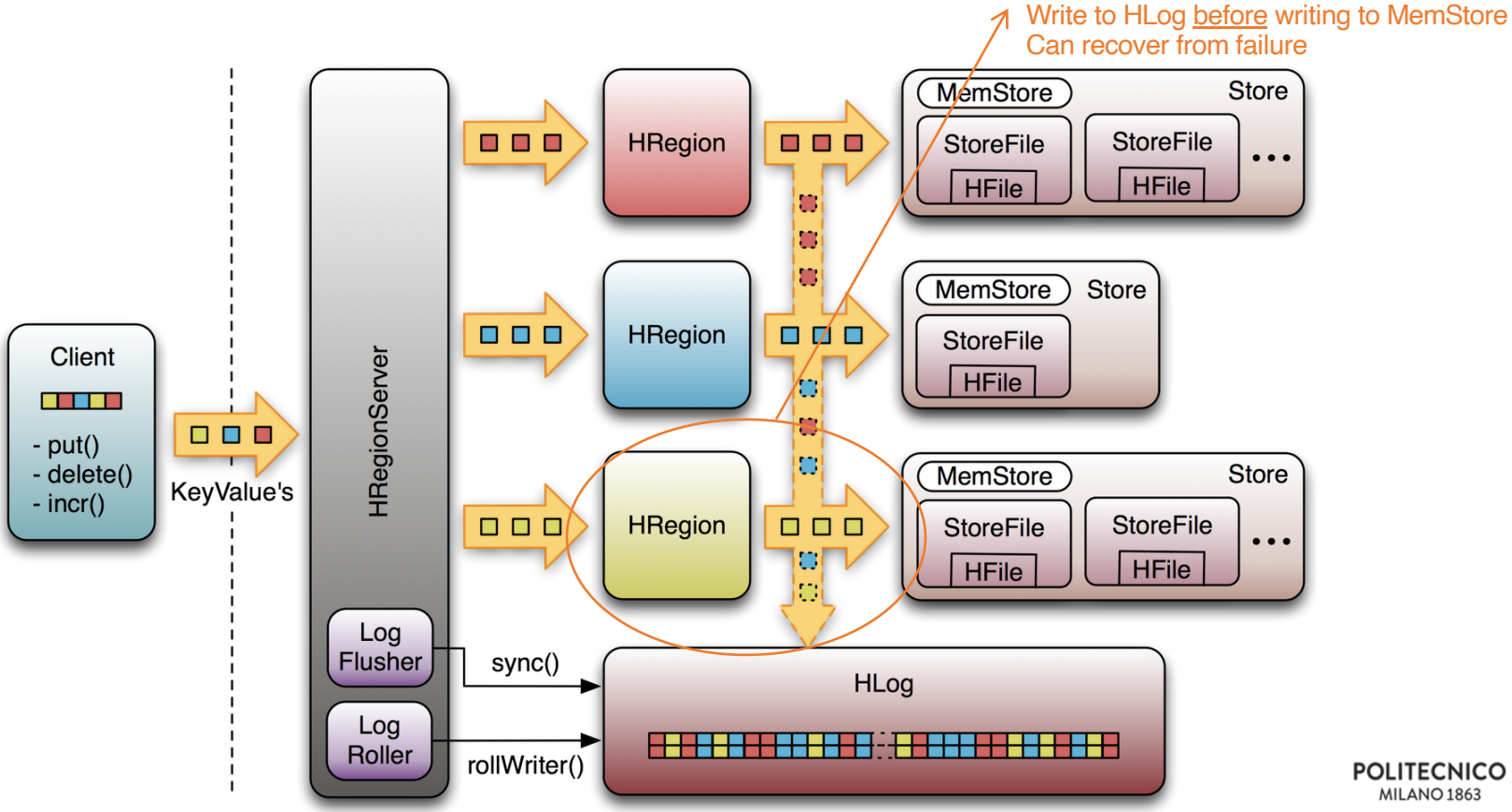
HBase Storage hierarchy

- HBase Table
  - Split it into multiple regions: replicated across servers
    - One Store per ColumnFamily (subset of columns with similar query patterns) per region
      - Memstore for each Store: in-memory updates to Store; flushed to disk when full
        - StoreFiles for each store for each region: where the data lives
          - Blocks
- HFile
  - SSTable from Google's BigTable

HFile


Data
Data
Data
Meta
(Optional)
Meta
(Optional)
File Info
Data
Index
Meta
Index
Trailer
Magic
Key
Value
Key
Value
Key
Value
Key
Value
Key
Value
Key
Value
(For a census table example)
Key
Key
Length
Value
Length
Row
Length
Row
Column
Family
Length
Column
Family
Column
Qualifier
Time
Stamp
Key
Type
Value
SSN:000-00-0000
Demographic
Ethnicity

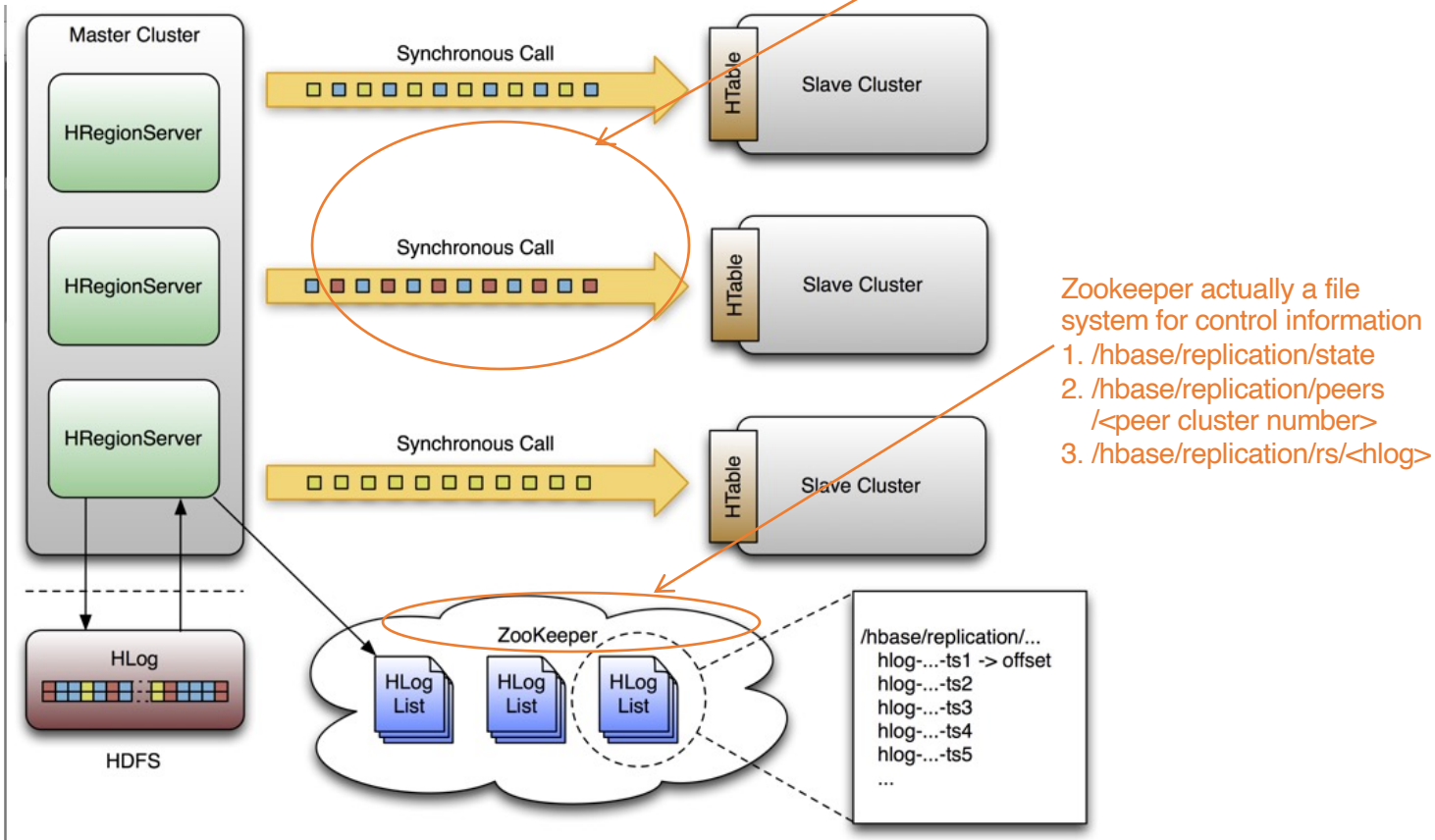
Strong Consistency: HBase Write-Ahead Log



Log Replay

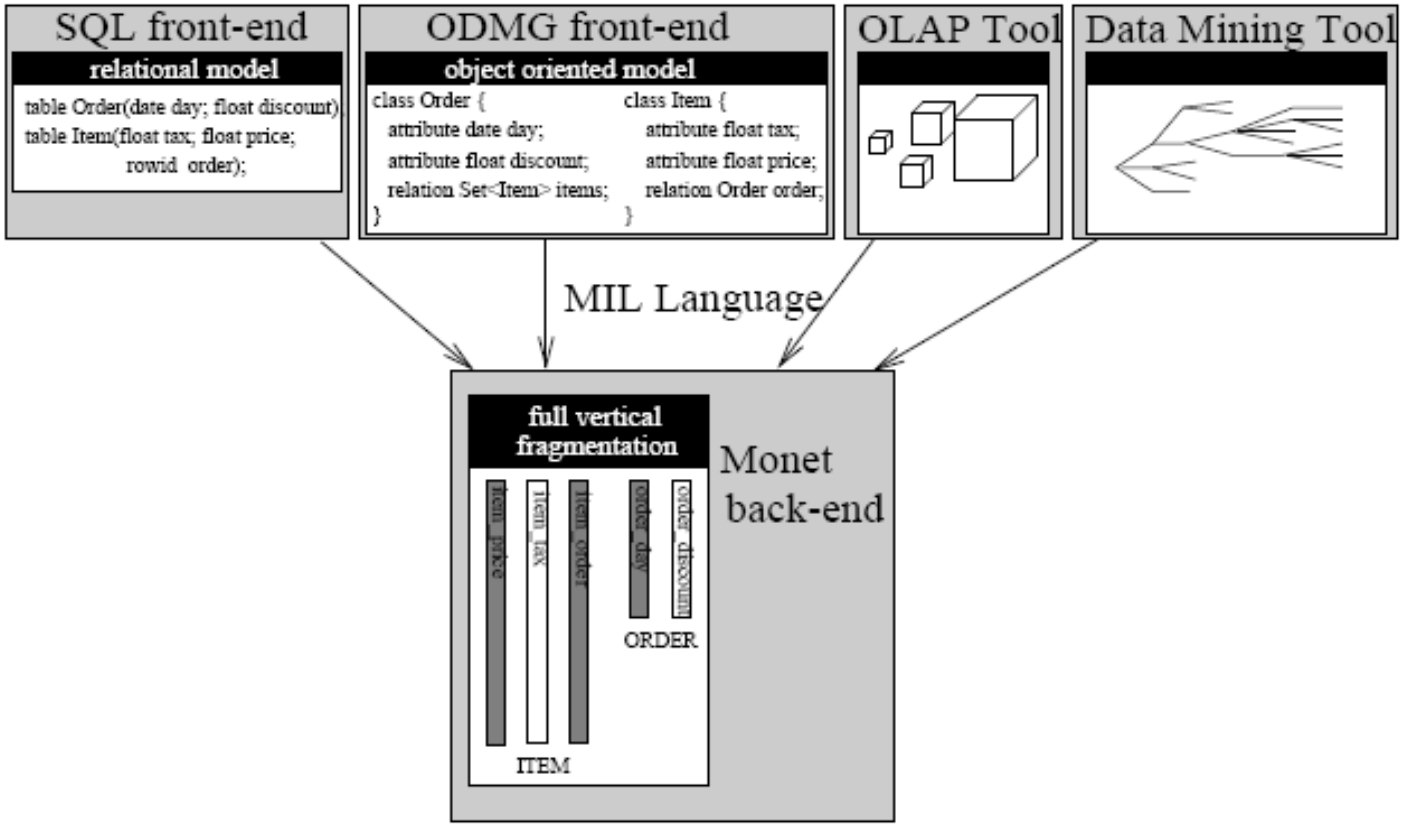
- After recovery from failure, or upon bootup (HRegionServer/HMaster)
  - Replay any stale logs (use timestamps to find out where the database is w.r.t. the logs)
  - Replay: add edits to the MemStore
- Why one HLog per HRegionServer rather than per region?
  - Avoids many concurrent writes, which on the local file system may involve many disk seeks

Cross-data center replication



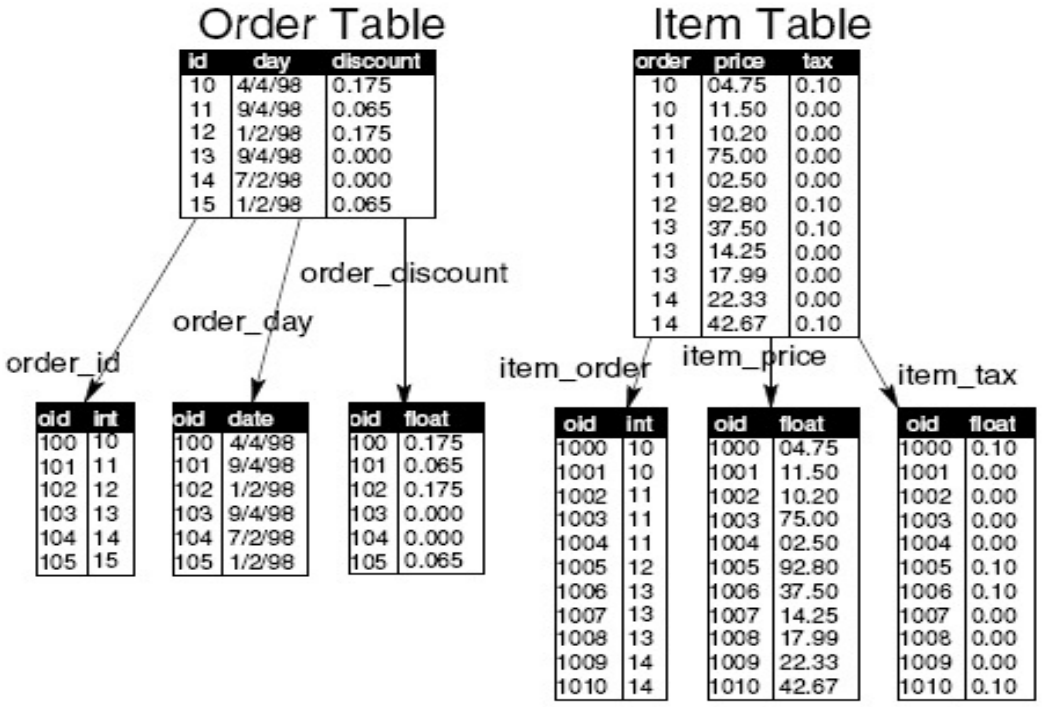
Other Columnar DBs

MonetDB


SQL front-end
relational model
table Order(date day; float discount)
table Item(float tax; float price;
rowid order);
ODMG front-end
object oriented model
class Order {
attribute date day;
attribute float discount;
relation Set<Item> items;
}
class Item {
attribute float tax;
attribute float price;
relation Order order;
}
OLAP Tool
Data Mining Tool
MIL Language
full vertical fragmentation
item_price
item_tax
item_order
order_day
order_discount
ITEM
ORDER
Monet back-end

MonetDB(contd)

```
table Order(int id; date day; float discount);
table Item(int order; float price; float tax);
```

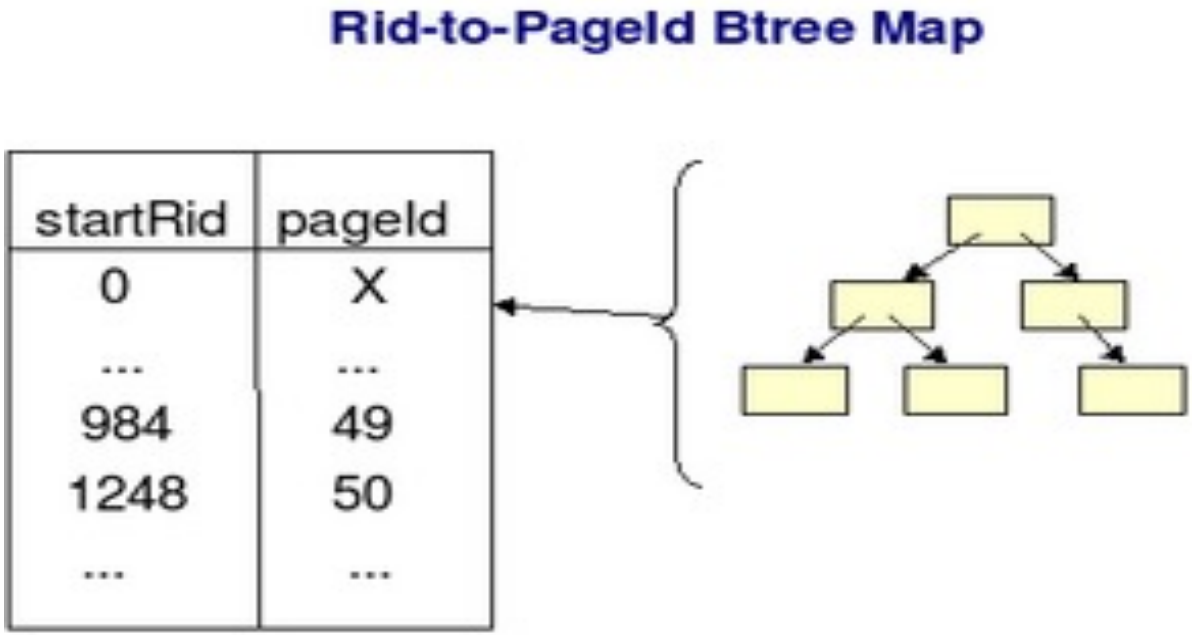


LucidDB

- LucidDb tables are column store tables
- Data in LucidDB is stored in Operating System in a file name as db.dat
- Column store table consists of set of clusters.
- Each column maps to single cluster.
- A single cluster page, therefore, stores the values for a specific set of rowIDs for all columns in that cluster.

- Each cluster also has associated with it a btree index.
- The btree index maps rid values to pageIds.
- The rid values correspond to the first rid value stored on each page within a cluster, and the cluster pages are identified by their pageIds.

LucidDB(contd)



LucidDB(contd)

- Within a cluster page, column values, by default, are stored in a compressed format, which allows LucidDB to minimize storage requirements.
- The idea here is instead of storing each column value for every rid value on a page, we instead store just the unique column values.
- We then associate with each column value a bit-encoded vector

Conclusion

- Column architecture doesn’t read unnecessary columns
- Avoids decompression costs and performs operations faster.
- Use of compression schemes allow us to lower our disk space requirements.

POLITECNICO
MILANO 1863

SYSTEMS AND METHODS FOR BIG AND UNSTRUCTURED DATA

Columnar Databases – Cassandra

Marco Brambilla

marco.brambilla@polimi.it

@marcobrambi