Data Engineering 2020 Fall

LTAT.02.007

Ass Prof. Riccardo Tommasini

Assistants: Fabiano Spiga, Mohamed Ragab, Hassan Eldeeb

https://courses.cs.ut.ee/2020/dataeng

Forum

Moodle

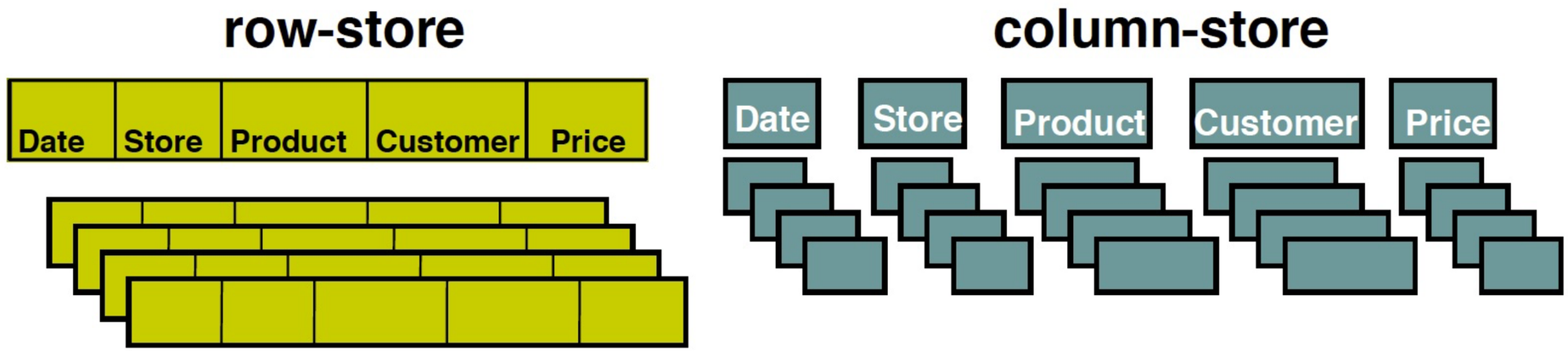
Column Oriented Database

The approach to store and process data by column instead of row has its origin in analytics and business intelligence

Column-stores operating in a **shared-nothing** massively parallel processing architecture can be used to build high-performance applications.

The class of column-oriented stores, which sees in Google's BigTable it's first member, is seen less puristic, also subsuming datastores that integrate column- and row-orientation.

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

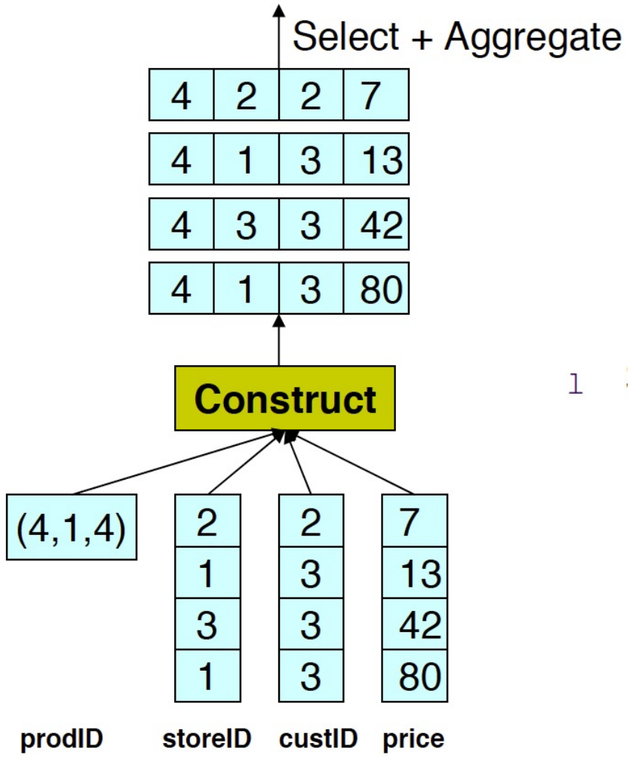
Pros and Cons

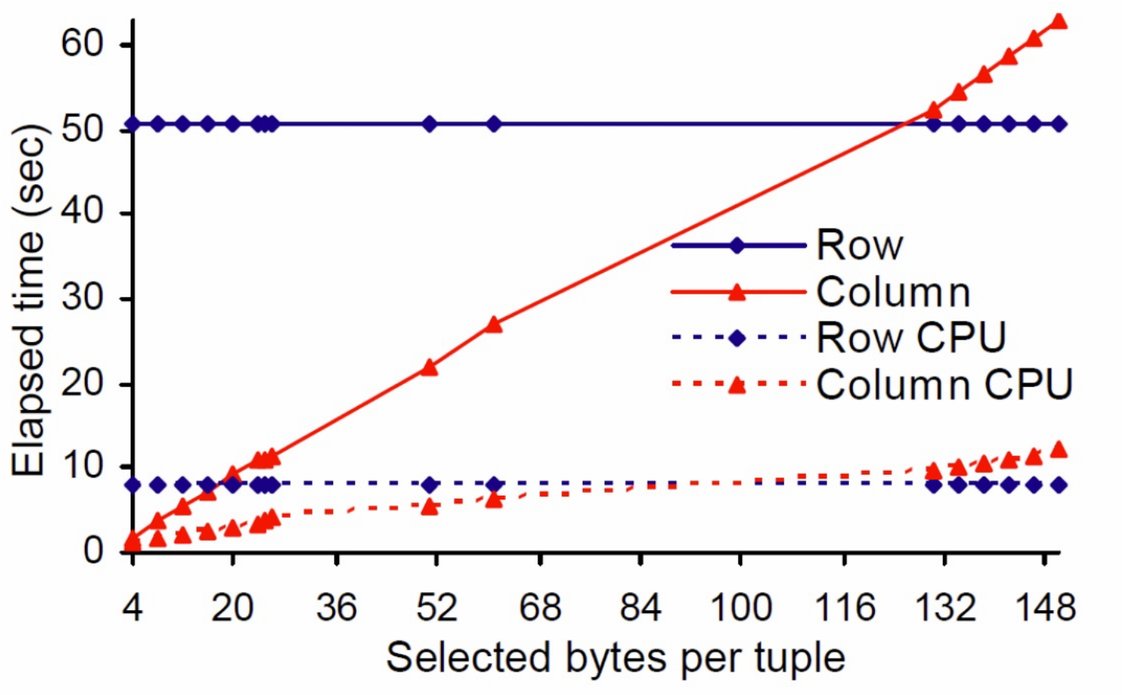
- Data compression
- Improved Bandwidth Utilization
- Improved Code Pipelining
- Improved cache locality

- Increased Disk Seek[70] Time
- Increased cost of Inserts
- Requires disk prefetching
- Adds tuple reconstruction costs

[70] Seek time is the time taken for a hard disk controller to locate a specific piece of stored data

Tuple Reconstruction





source

Compression

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

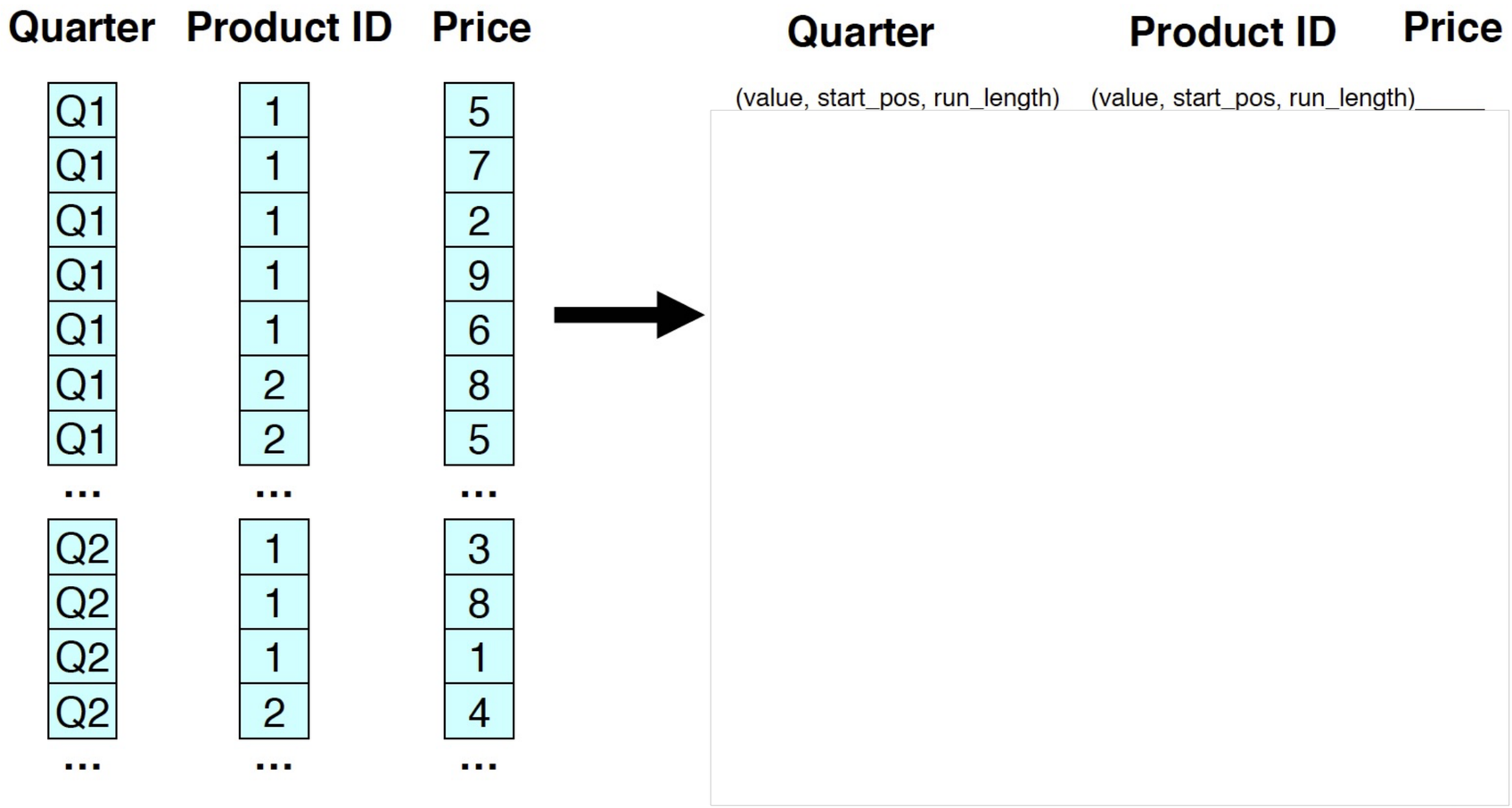
Extra Avaiable  Paper Summary

Example (String): Run-Length Encoding

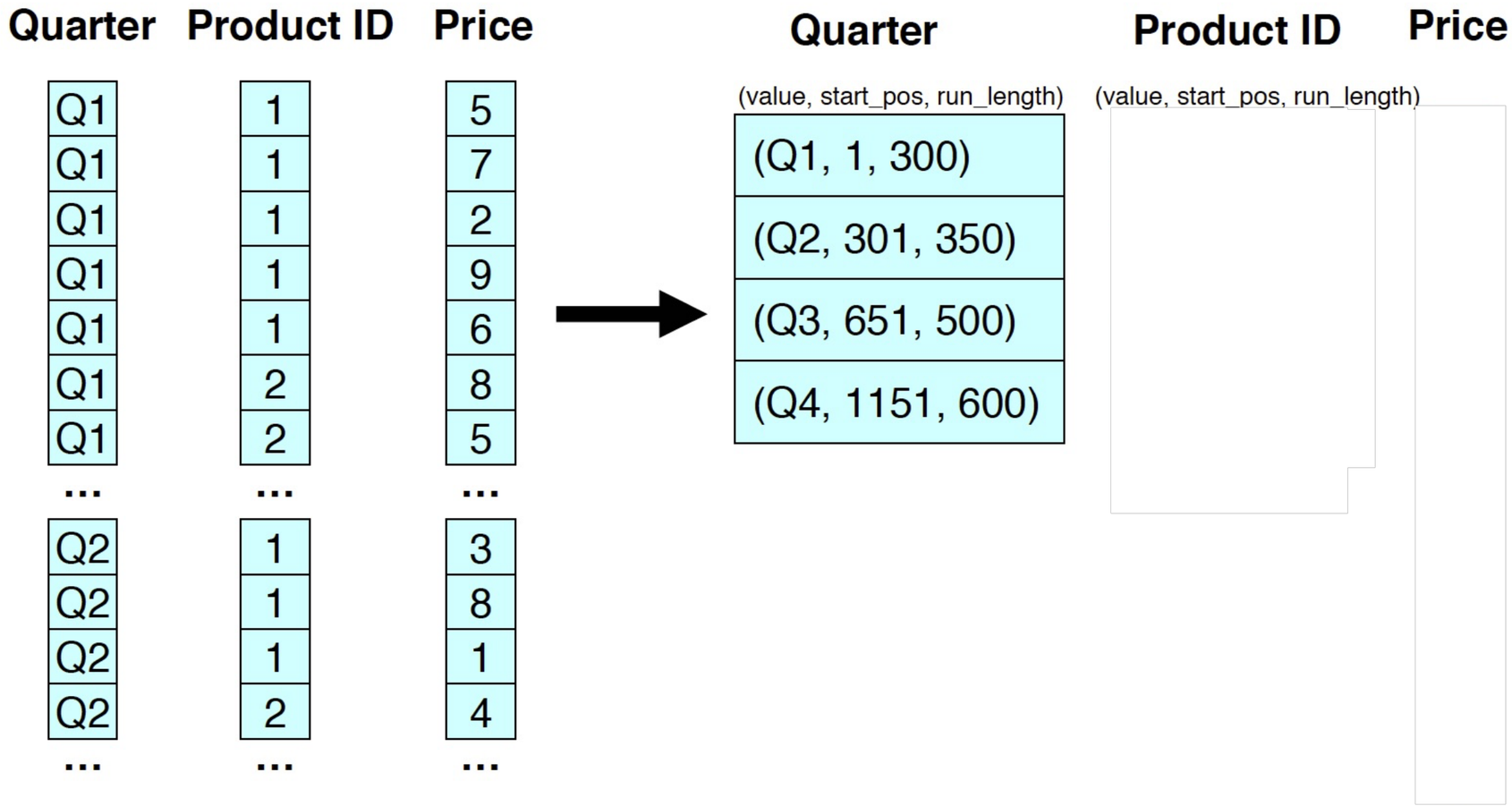
aaaabbbbbbbbbbbbccccccccddddddddeeeeeeddddd

4a13b7c7d5e5f

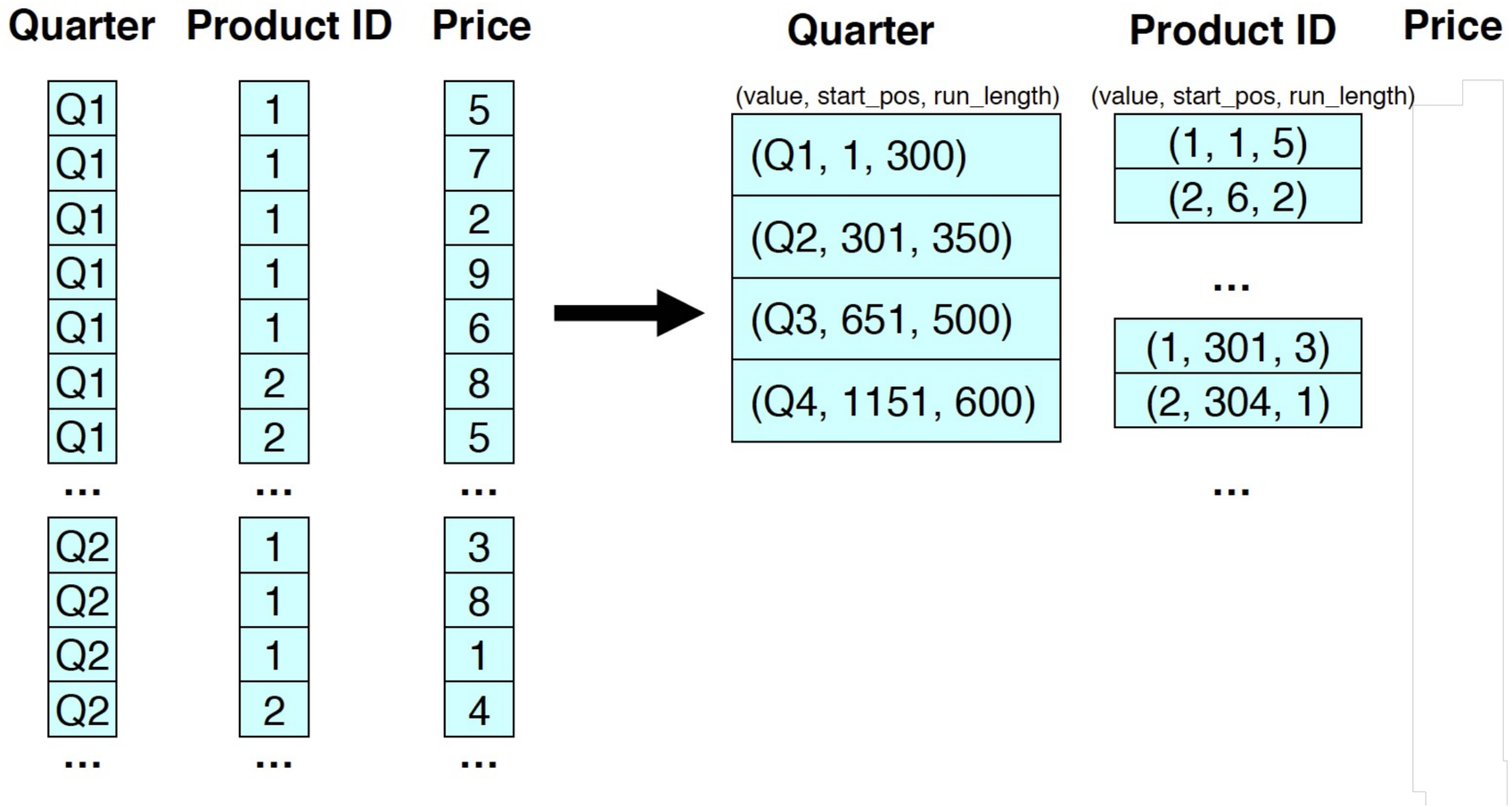
Example (DB): Run-Length Encoding



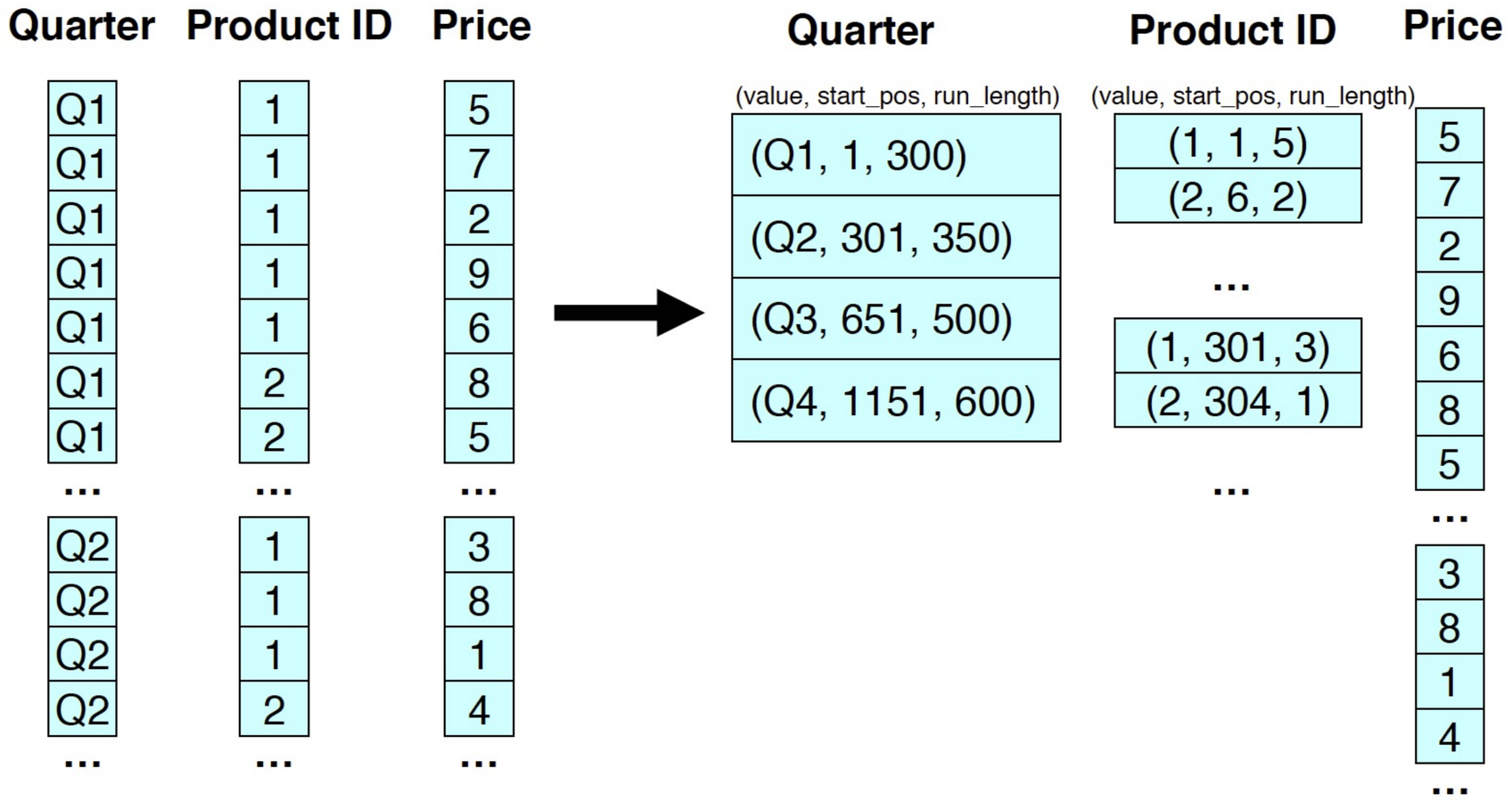
Compression: Run-Length Encoding



Compression: Run-Length Encoding



Compression: Run-Length Encoding



List of Databases

- **[[Cassandra]]**
- Vertica
- SybaseIQ
- C-Store
- BigTable/[[HBASE]]
- MonetDB
- LucidDB

Cassandra

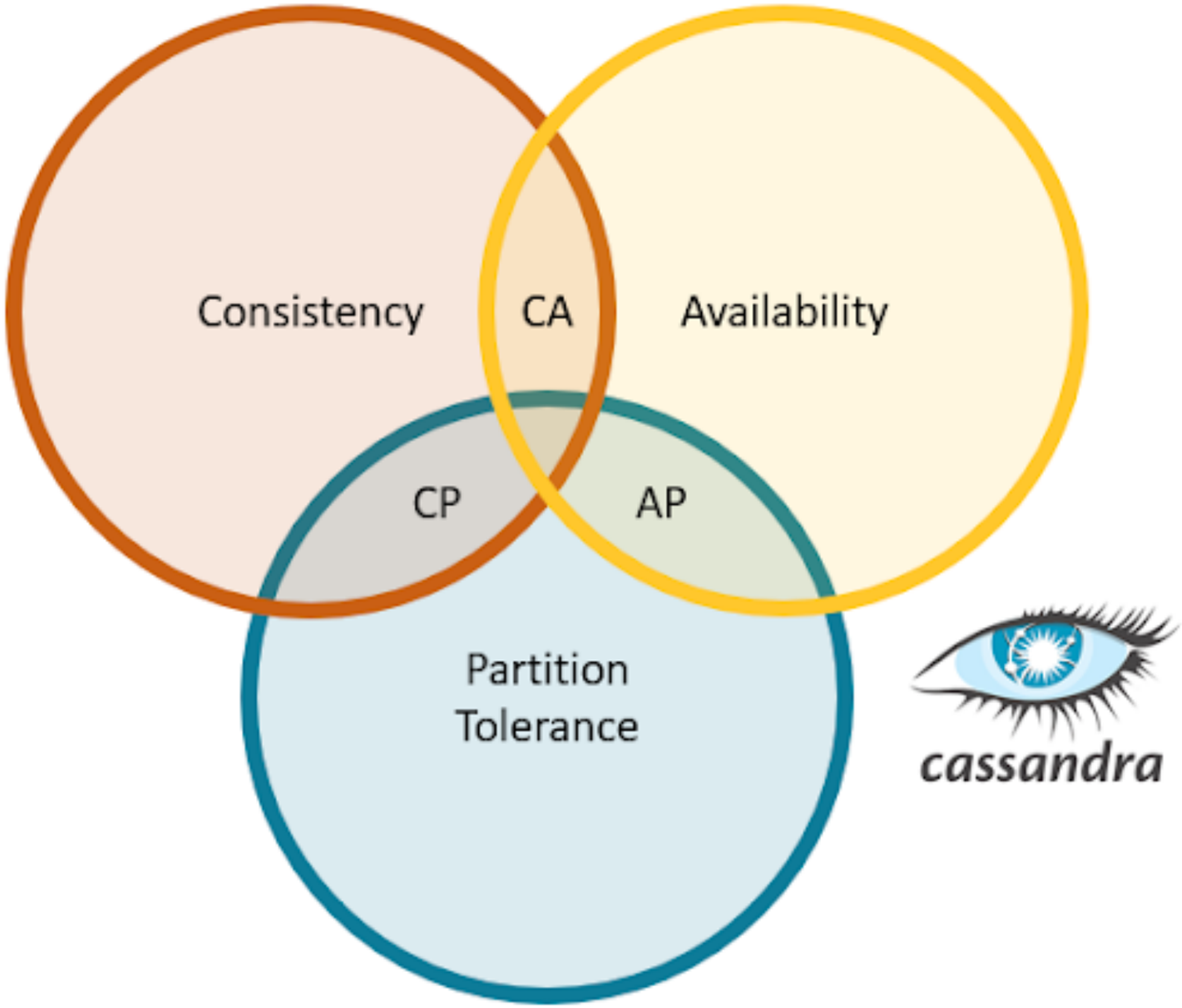
History of Cassandra

Originally designed at Facebook

Open-sourced and now within Apache foundation

What Cassandra is

- A Wide [[Column Oriented Database]]
- *tuneably* consistent (~~**C**~~)
- very fast in writes
- highly available (**A**)
- fault tolerant (**P**)
- linearly scalable, elastic scalability
- Cassandra is very good at writes, okay with reads.



What Cassandra is not

- Cassandra is not a replacement for Relational Databases
- Tables should **not** have multiple access paths
- Cassandra does not support aggregates, if you need to do a lot of them, think another database.
- Updates and deletes are implemented as special cases of writes and that has consequences that are not immediately obvious.

Comparison with RDBMS

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

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.

Use Cases

The use-case leading to the initial design and development of Cassandra was the so entitled Inbox Search problem at Facebook.

-  Purchases, test scores
- Storing time series data (as long as you do your own aggregates).
  - Storing health tracker data.
  - Weather service history.
  -  User Activity
- Internet of things status and event history.
-  IOT for cars and trucks
- Email envelopes—not the contents.

When to consider Cassandra

- you need really fast writes
- you need durability
- you have lots of data (> GBs) and (>=) three servers
- your app is evolving
  - startup mode, fluid data structure
- loose domain data
  - “points of interest”
- your programmers can handle
  - complexity
  - consistency model
  - change
  - visibility tools
- your operations can deal
  - hardware considerations
  - data transport
  - JMX monitoring

Advantages

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 Model

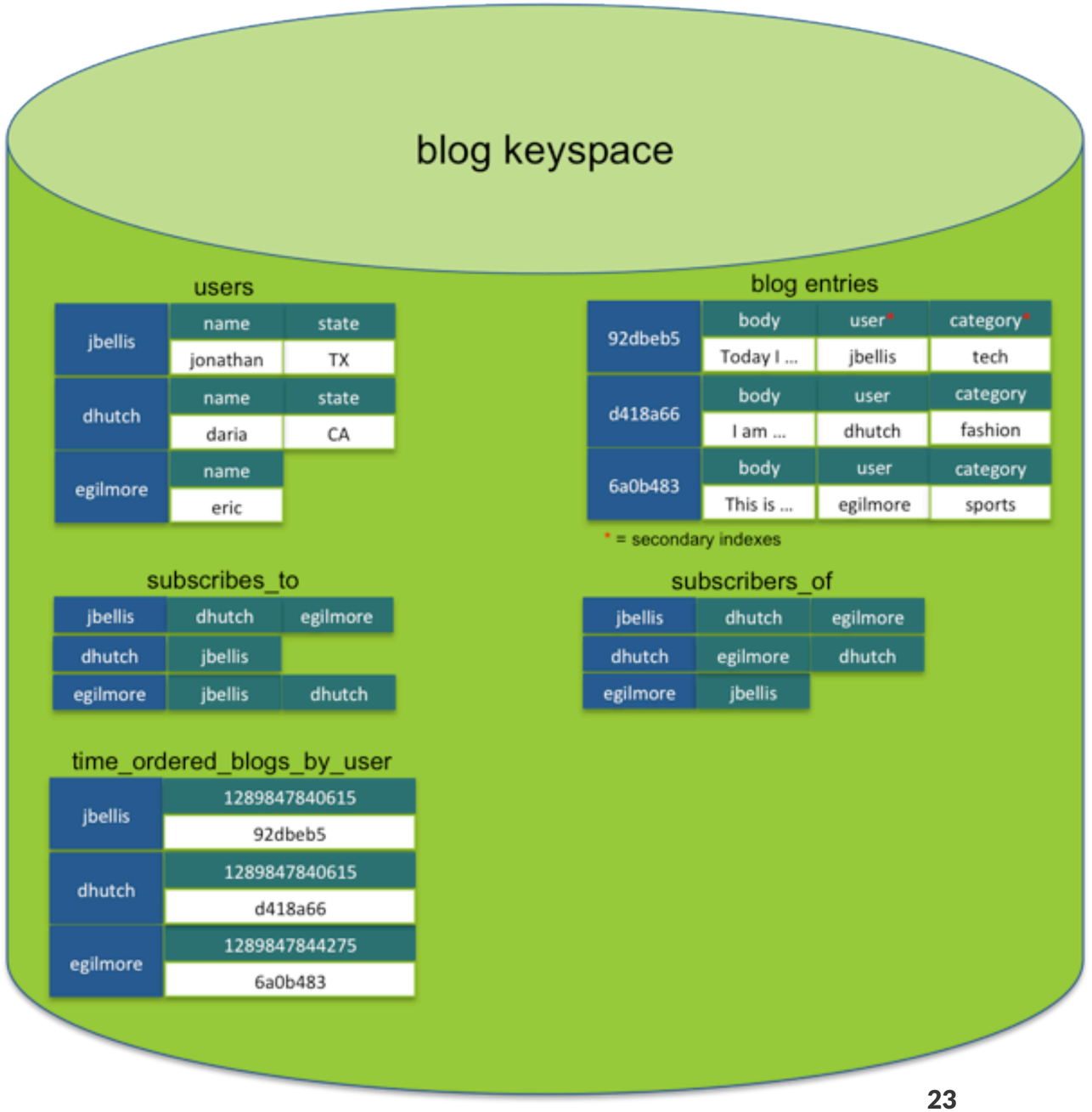
RDBMSs: domain-based model
-> what answers do I have?

Cassandra: query-based model
-> what questions do I have?

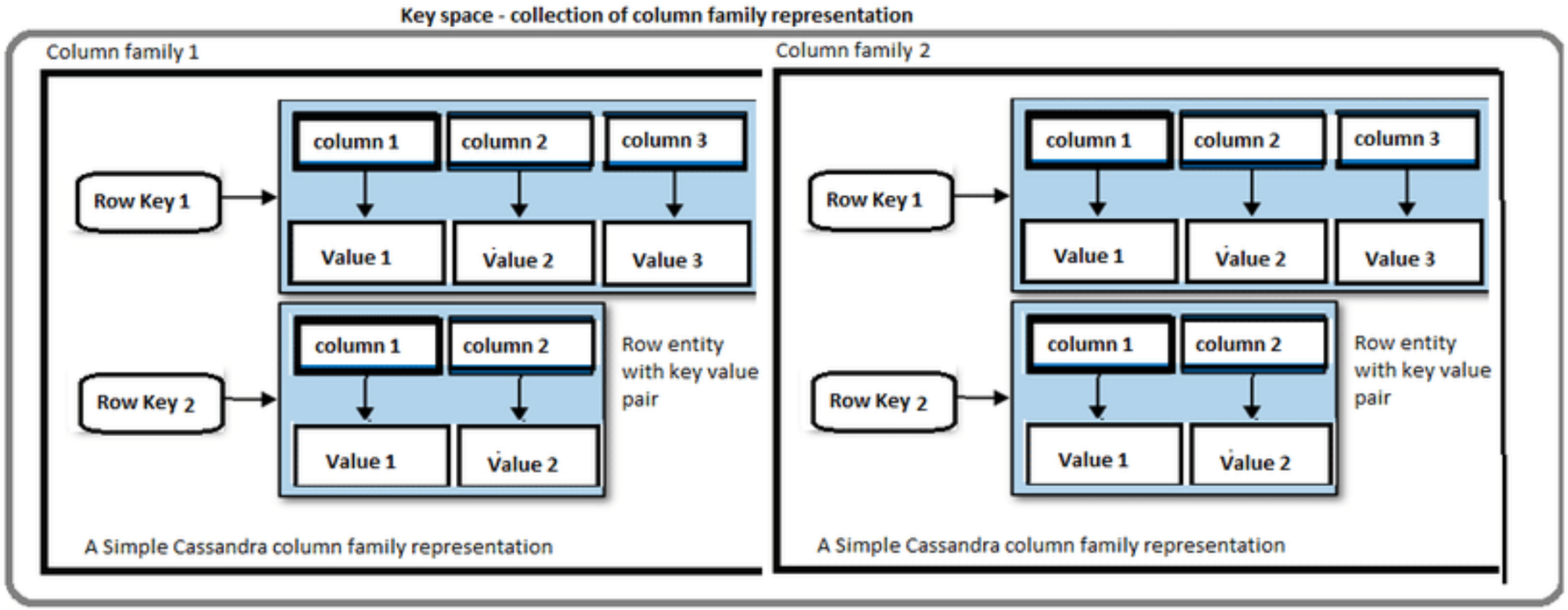
Cassandra does **not** support a full relational data model.

Instead, it provides clients with a simple data model that supports **dynamic control** over data layout and formats.

An instance of Cassandra typically consists of a one distributed multidimensional map indexed by key which contains one or more **column families** that, in turn, **rows**

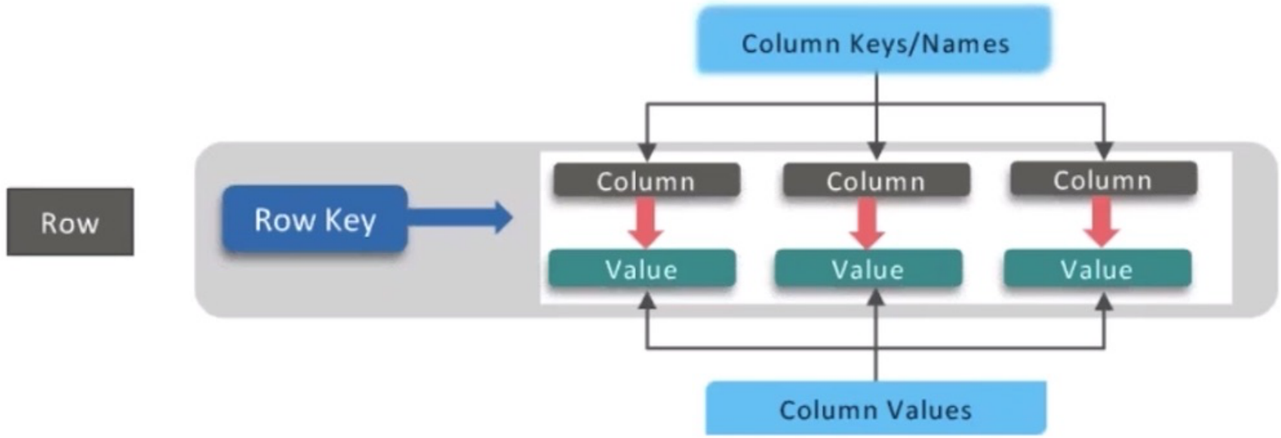


- **Rows** are identified by a string-key
- **Column Families** correponds to tables in RDBMS but may be unstructured. A column family consists of
  - **(Simple) Columns Families** have a name and store a number of values per row which are identified by a timestamp
  - **Super Columns Families** have a name and an arbitrary number of columns associated with them



Keyspace

- Key space is typically one per application
- Keys are similar to those of databases
- Some settings are configurable only per keyspace
- Each Row must have a key



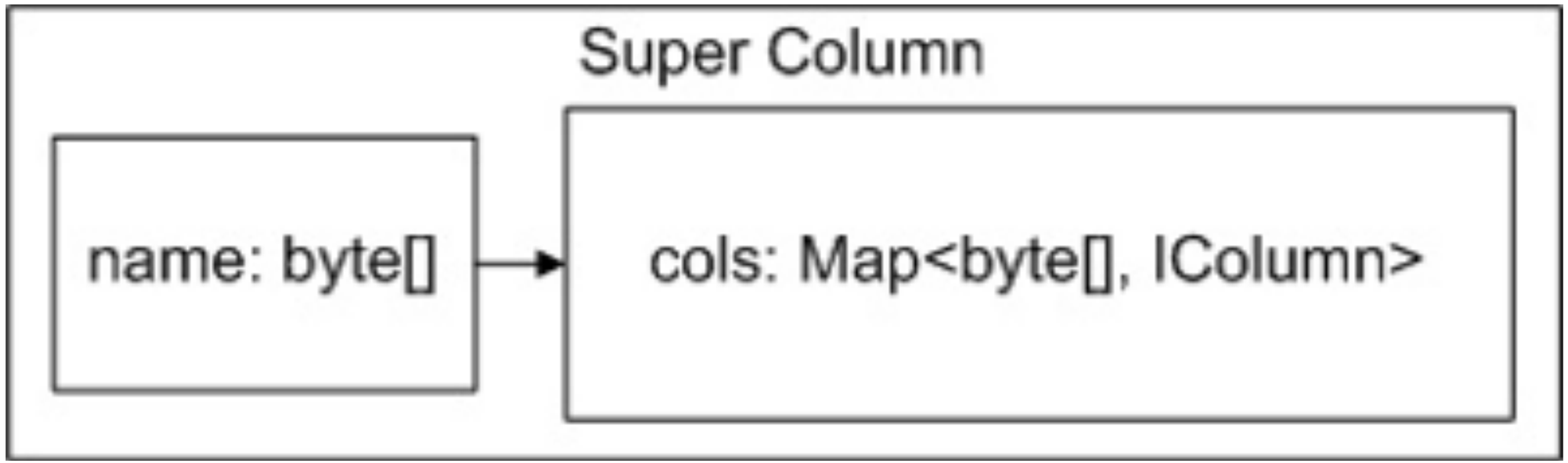
Columns Families

A Column consists of three parts

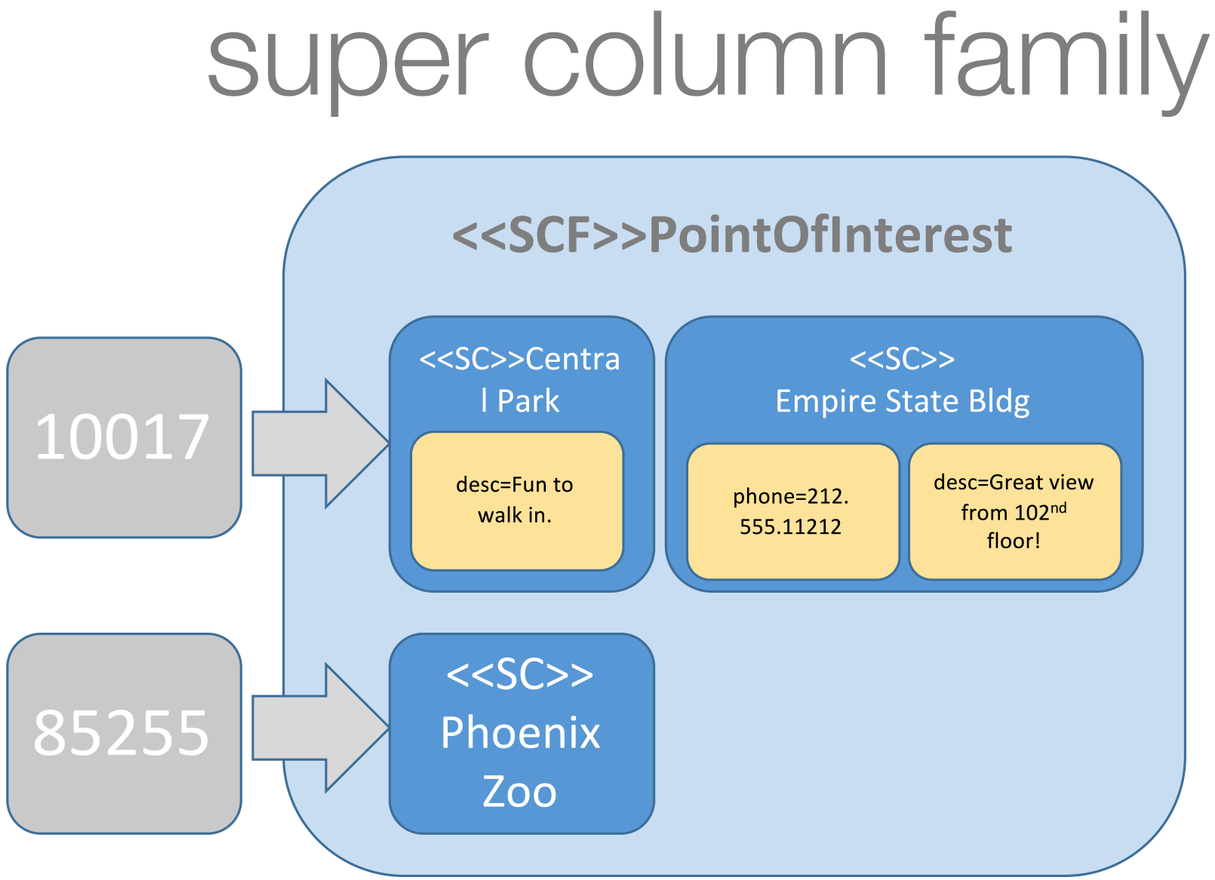
- name
  - byte[]
  - determines sort order
  - used in queries
  - indexed
- value
  - byte[]
  - you don’t query on column values
- timestamp
  - long (clock)
  - last write wins conflict resolution

Super Column Families

- Super columns group columns under a common name
- sub-column names in a Super Column Family are **not** indexed
  - top level columns (Super Column Family Name) are **always** indexed
- often used for **denormalizing** data from standard Column Families



Example


super column family
<<SCF>>PointOfInterest
10017
<<SC>>Centra
l Park
desc=Fun to
walk in.
<<SC>>
Empire State Bldg
phone=212.
555.11212
desc=Great view
from 102nd
floor!
85255
<<SC>>
Phoenix
Zoo

Example (Json Notation)

```
 PointOfInterest { //Supercolumn Family
  key:85255 {
     Phoenixzoo { phone: 480-555-5555, //column
          desc: They have animals here //column },
     Spring Training {
          phone: 623-333-3333,  //column
            desc: Fun for baseball fans. //column
    }
  } //end phoenix,

  key: 10019 {
     Central Park   //super column
        { desc: Walk around. It's pretty. // missing phone column } ,
      Empire State Building { phone: 212-777-7777,
             desc: Great view from 102nd floor. }
    } //end nyc
}
```

Architecture

Cassandra is required to be incrementally scalable.

Therefore machines can join and leave a cluster (or they may crash).

Data have to be **partitioned** and **distributed** among the nodes of a cluster in a fashion that allows *repartitioning* and *redistribution*.

Partitioning

- Data of a Cassandra table get partitioned and distributed among the nodes by a consistent **order-preserving** hashing function.
- The order preservation property of the hash function is important to support **range scans** over the data of a table.
- Cassandra performs a **deterministic** load balancing
  - it measures and analyzes the load information of servers and moves nodes on the consistent hash ring to get the data and processing load balanced.

Replication

- Data get replicated to a number of nodes which can be defined as a **replication factor** per Cassandra instance.
- Replication is managed by a **coordinator node** for the particular **key** being modified.
- The coordinator node for any key is the **first node on the consistent hash ring** that is visited when walking from the key's position on the ring in **clockwise** direction.

~~Replication Strategies~~ (Used to be)

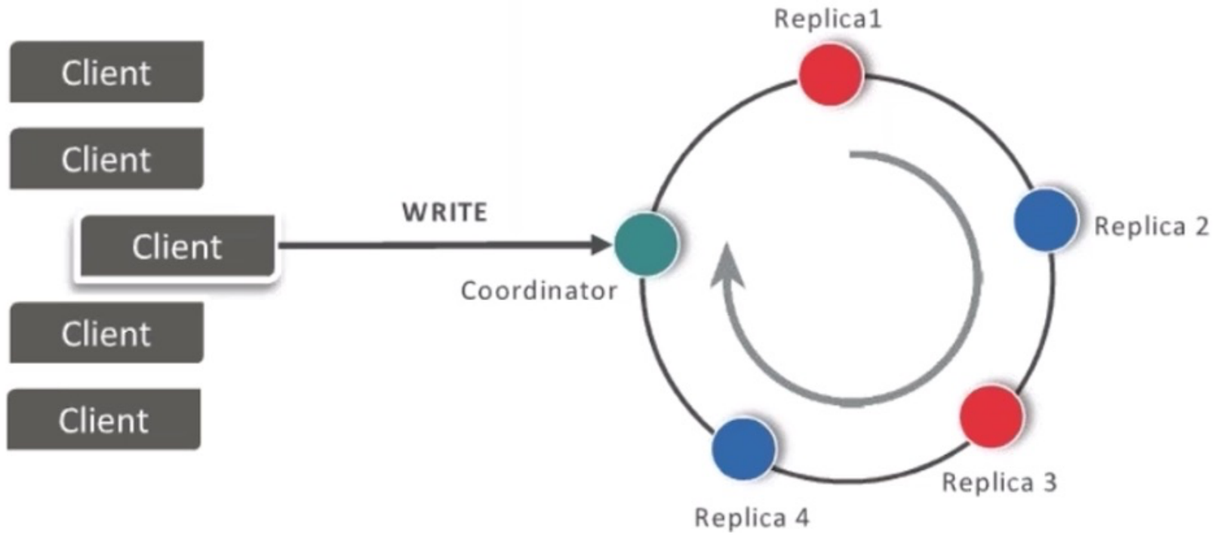
- **Rack Unaware**: the non-coordinator replicas are chosen by picking N-1 successors of the coordinator on the ring
- **Rack Aware** and **Datacenter Aware** rely on Zookeeper for leader election.
  - the elected leader is in charge of maintaining the invariant that no node is responsible for more than N-1 ranges in the ring

Replica placement strategies Today [81]

[81] docs

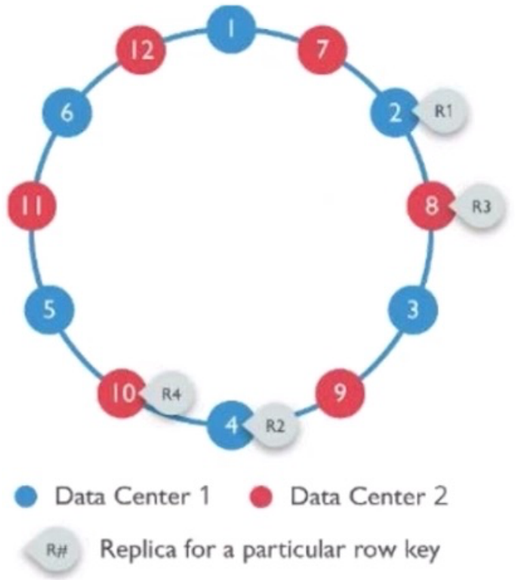
Simple Strategy

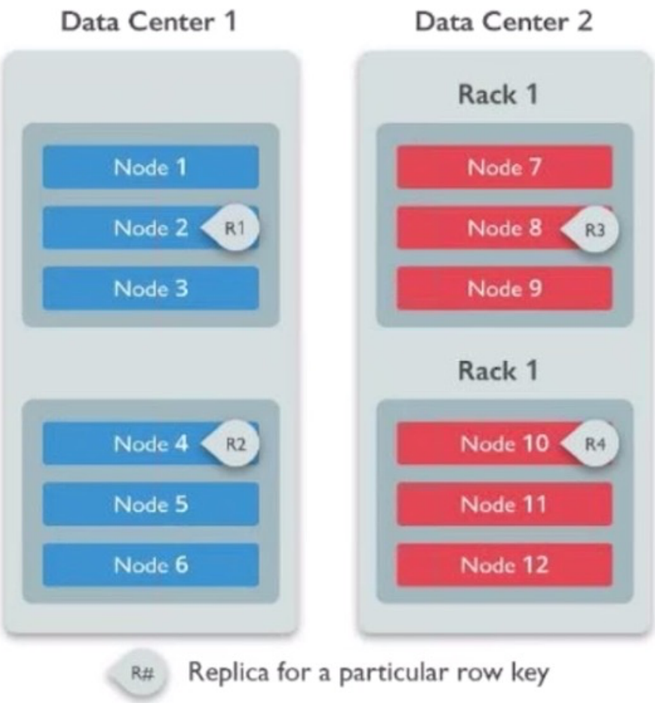
- Allows a single integer *replication_factor* to be defined
- Single datacenter
- Clockwise placement to the next node(s)
- all nodes are treaded equally, ignoring any configured data centers or racks.



Network Topology Strategy

- Multiple datacenters
- Allows a single integer *replication_factor* to be defined per data center
- Attempts to choose replicas within a data center from different racks as specified by the Snitch[82]
- Supports local (reads) queries





[82] Snitch teaches Cassandra about your network topology to route requests efficiently.

Partitioner Smack-Down

Random Preserving

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

Order Preserving

- key distribution determined by token
- lexicographical ordering
- required for range queries
- can specify the token for this node to use

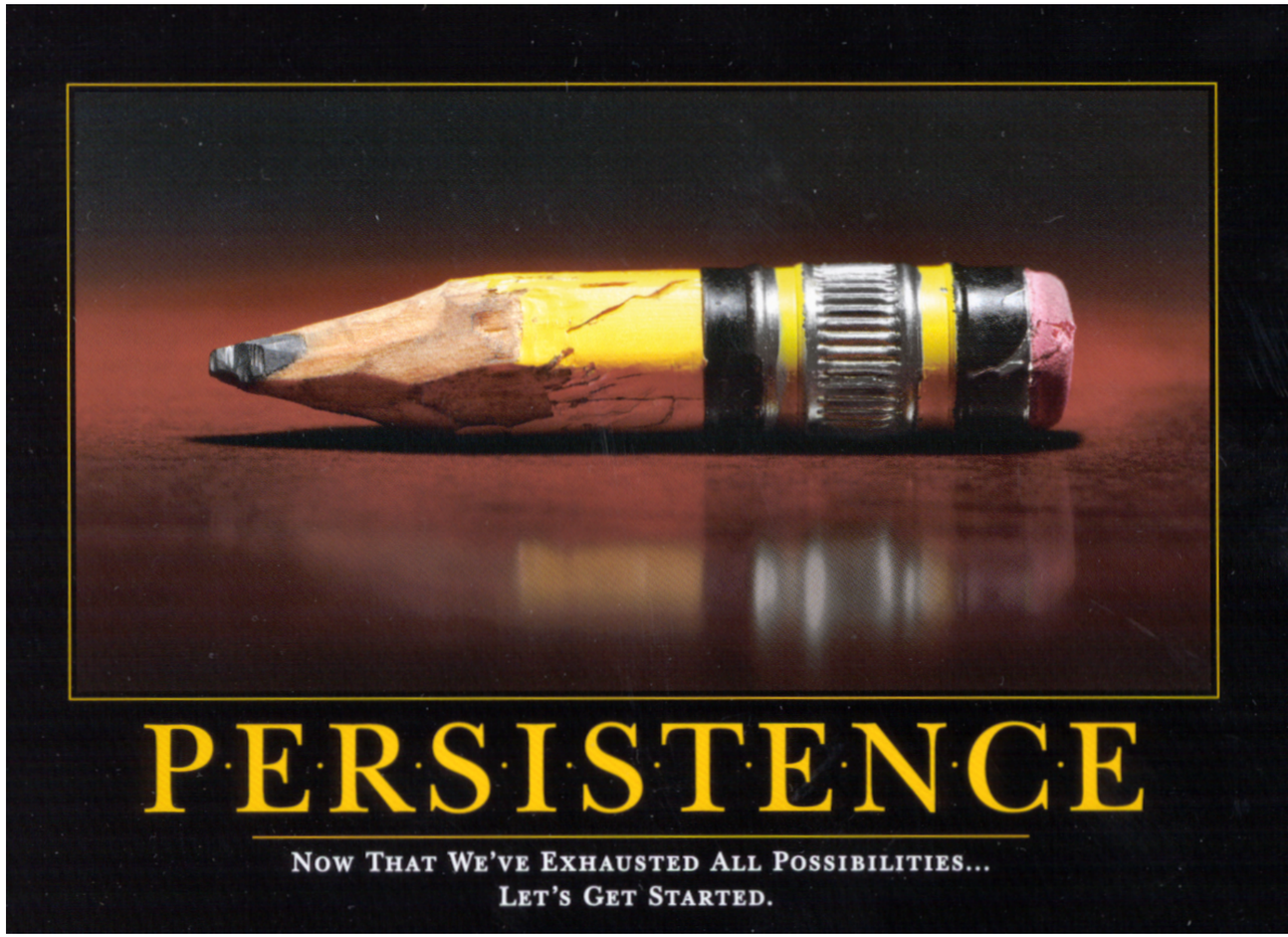
Persistence

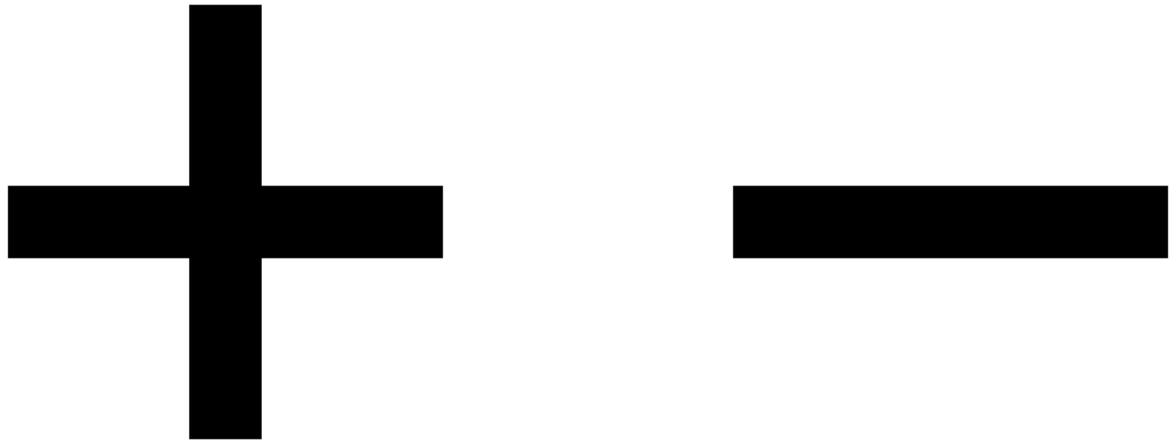
Cassandra provides **durability guarantees** in the presence of node failures and network partitions by relaxing the quorum requirements

The Cassandra system relies on the **local file system** for data persistence.

The data is **represented** on disk using a format that lends itself to **efficient** data **retrieval**.

Typical write operation involves a write into a **commit** log for durability
and recoverability and an update into an in-memory data structure.





Operations

Writes

- Need to be lock-free and fast (no reads or disk seeks)
- AClient sends write to one front-end node in Cassandra cluster (Coordinator)
- Coordinator sends it to all replica nodes responsible for that key
- A write is atomic at the partition-level, meaning inserting columns in a row is treated as one write operation.

Hinted Handoff

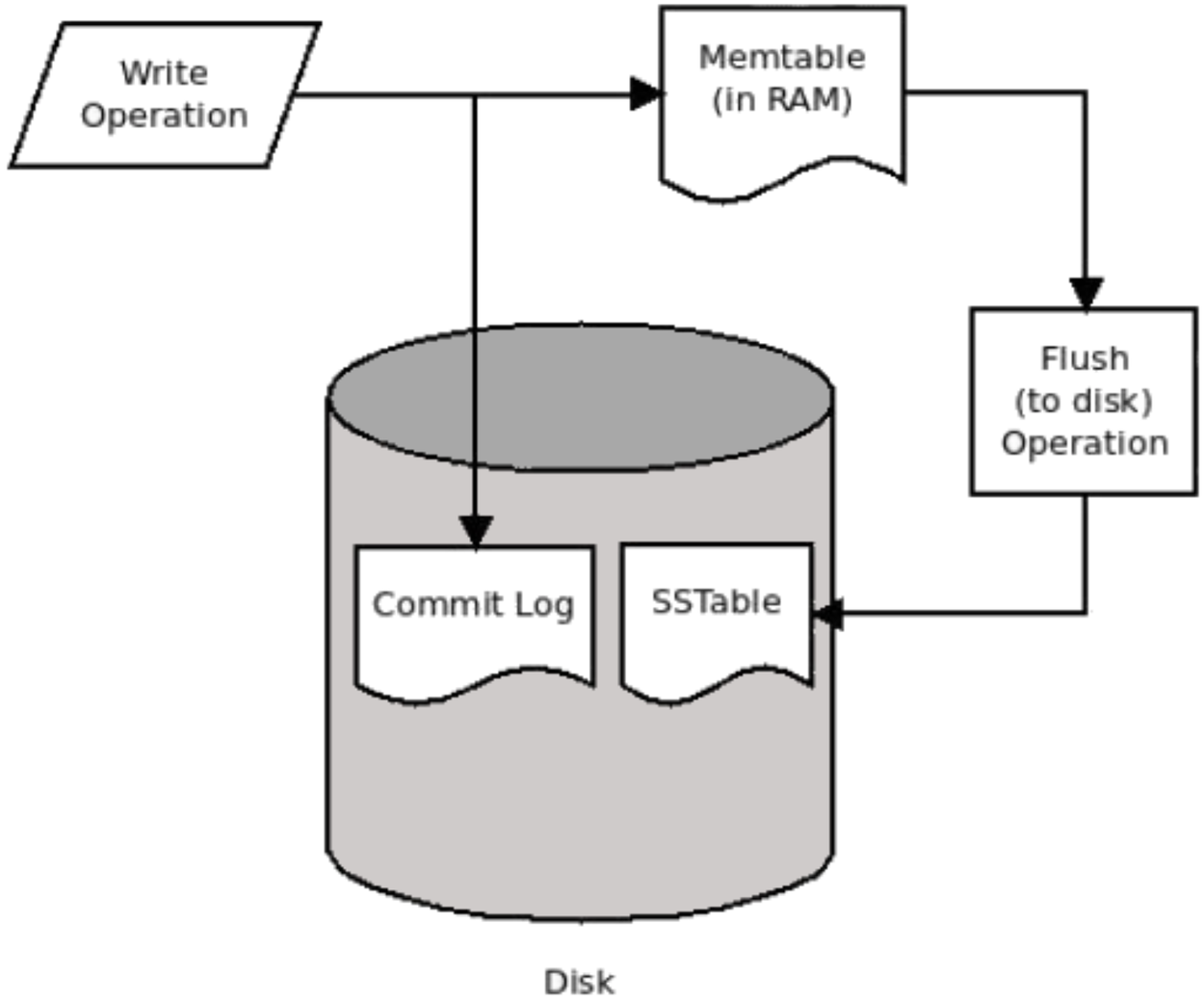
If any replica is down, the coordinator writes to all other replicas, and keeps the write until down replica comes back up.

When all replicas are down, the Coordinator (front end) buffers writes (for up to an hour).

Writing Flow

1. Cassandra logs it in disk commit log (disk)
2. Adds *values* to appropriate *memtables* [83]
3. When memtable is full or old, flush to disk using a Sorted String Table

source



[83] In-memory representation of multiple key-value pairs

Consistency levels for a write operations[87]

- 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 datacenter
- EACH-QUORUM: quorum in every datacenter
- ALL: all replicas all datacenters

---

[87] detailed discussion

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

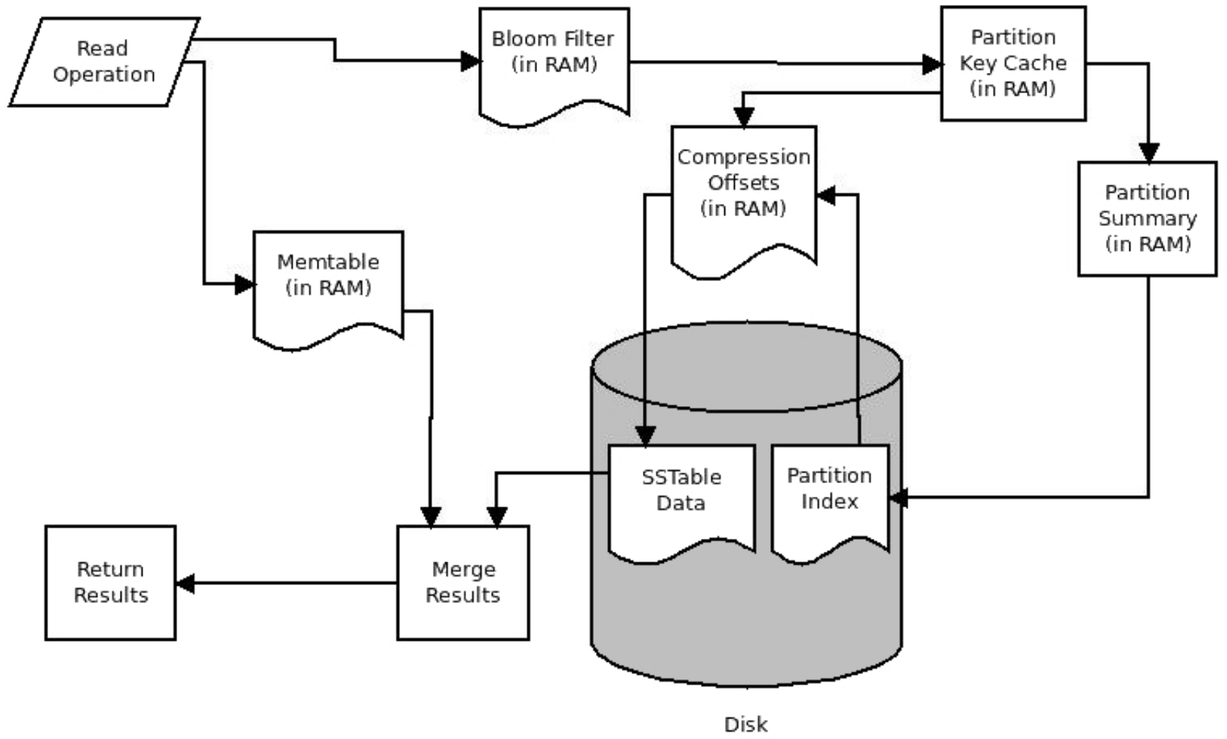
Reads

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

Reading Flow

1. Check row cache, if enabled
2. Checks partition key cache, if enabled
3. Check the memtable
4. Fetches the data from the SSTable on disk
5. If Row cache is enabled the data is added to the row cache

source



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

Read-Repair[84]

Read-repair is a **lazy** mechanism in Cassandra that ensures that the data you request from the database is accurate and consistent.

For every read request, the coordinator node requests to all the node having the data requested by the client. All nodes return the data which client requested for.

The most recent data is sent to the client and asynchronously, the coordinator identifies any replicas that return obsolete data and issues a read-repair request to each of these replicas to update their data based on the latest data.

---

[84] source

Consistency Level: Quorum

- N is replication factor
- R is read replica count,
- W is write replica count

- Quorum Q = N/2 + 1
- If W+R > N and W > N/2, you have consistency
- Allowed:

W	R
1	N
N	1
Q	Q

Consistency Level: Explained

Reads

- Wait for R replicas (R specified by clients)
- In background check for consistency of remaining N-R replicas

Writes

- **Block** until quorum is reached
- **Async**: Write to any node

Deletes

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


Digression Time

The Data Structure That Power Your Database[85]

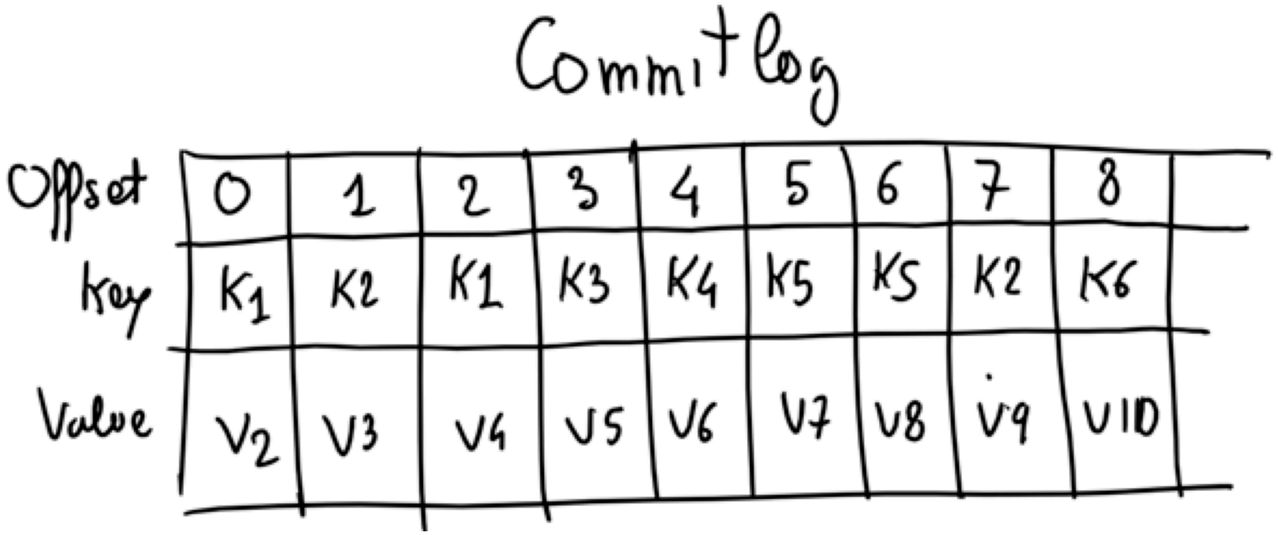
[85] Chapter 3 - Designing Data Intensive Applications

[[Log]]

A log is an append-only sequence of records. It doesn't have to be human-readable;

log-structured storage segments are typically a sequence of key-value pairs.

These pairs appear in the order that they were written, and values later in the log take precedence over values for the same key earlier in the log.



[Sorted String Table]

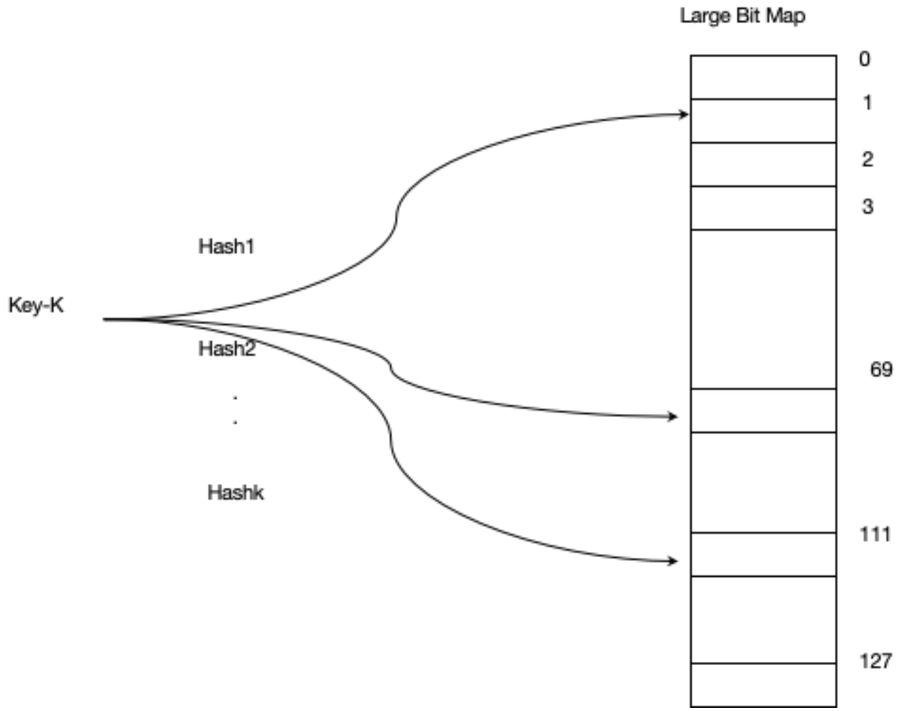
Make a simple change to logs: sequence of key-value pairs is sorted by key.

Merging segments is simple and efficient, even if the files are bigger than the available memory (mergesort algorithm).

In order to find a particular key in the file, you no longer just need a spare index of the offsets

[[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 maycheck 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%

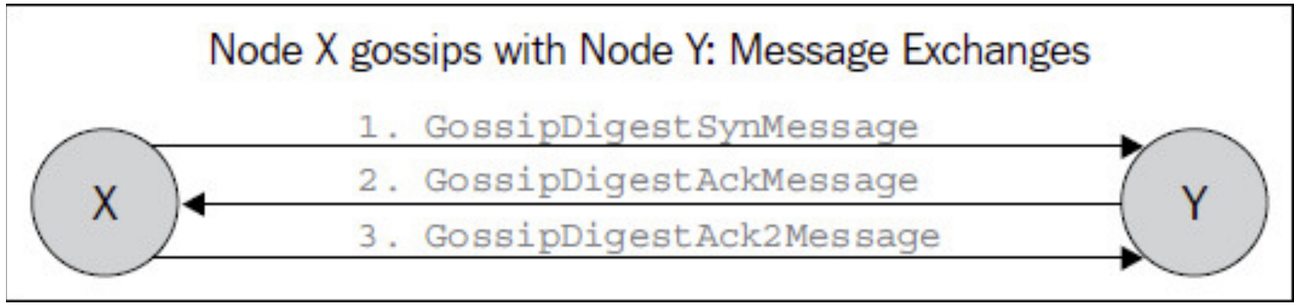
</Digression Time>

Cluster Membership

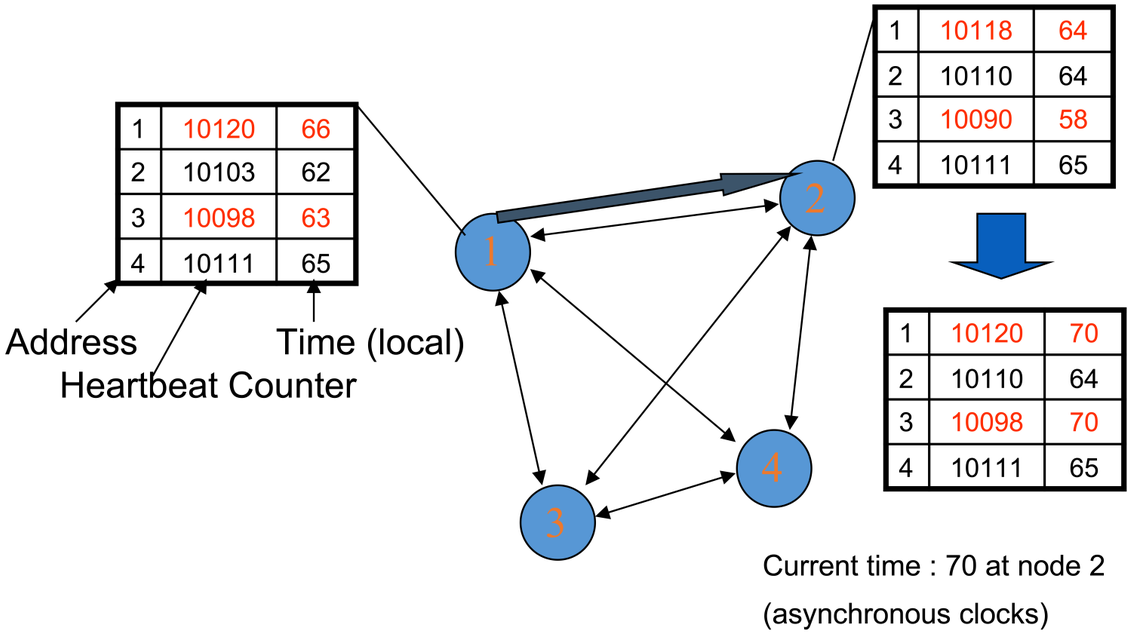
- Any server in cluster could be the coordinator
- So every server needs to maintain a list of all the other servers that are currently in the server
- List needs to be updated automatically as servers join, leave, and fail

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)



Gossip Protocol in practice



- regulates cluster membership
- Nodes periodically gossip their membership list
- On receipt, the local membership list is updated

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(tnow – tlast))/log 10
  - PHI basically determines the detection timeout, but is sensitive to actual inter-arrival time variations for gossiped heartbeats

Queries

Values in Cassandra are addressed by the triple (row-key, column-key, timestamp) with column- key as

- column-family:column (for simple columns contained in the column family)
- column-family: supercolumn:column (for columns subsumed under a supercolumn).

what about… CQL?

SELECT WHERE
ORDER BY
JOIN ON
GROUP

SELECT WHERE

Column Family: USER
Key: UserID
Colunms: username, email, birth date, city, state

How to support this query?

```
SELECT * FROM User WHERE city = ‘Scottsdae’
```

Create a new columns family called **UserCity**:

Column Family: USERCITY
Key: city
Colunms: 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

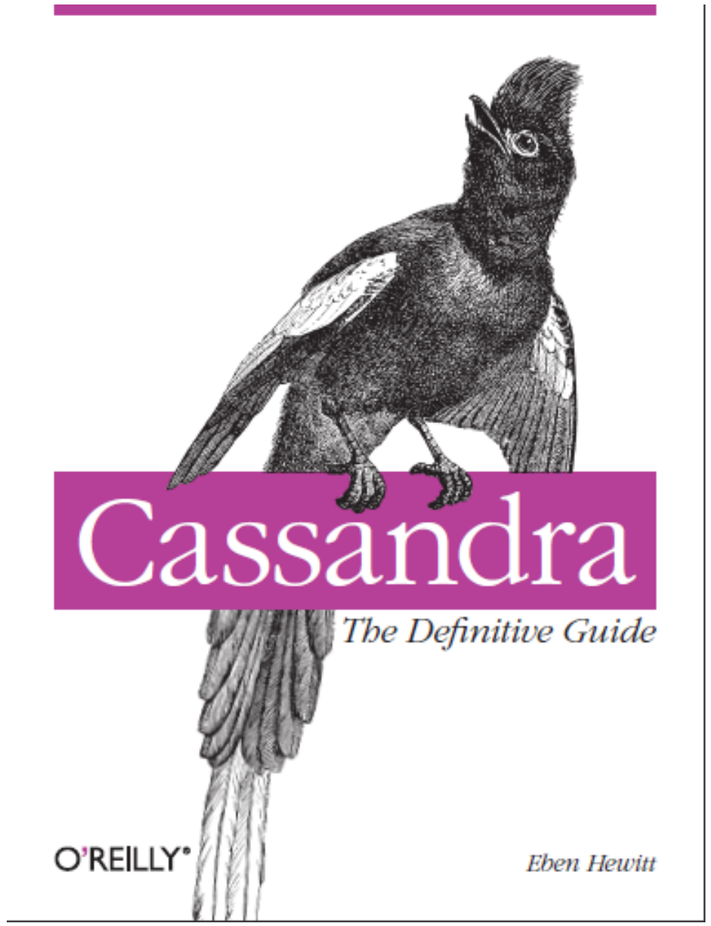
Columns

are sorted according to `CompareWith` or `CompareSubcolumnsWith`

Rows

- are *sorted* by key, regardless of partitioner
- are *placed* according to their Partitioner:
  - Random: MD5 of key
  - Order-Preserving: actual key

References



Extra (10)

Prepare Cassandra Practice

hints