

# Distributed Algorithms

## Cassandra

ד"ר מרים אללוף

# References

- ❑ H. Garcia-Molina, J. D. Ullman, J. Widom, Database Systems: The Complete Book. Second Edition. Pearson Prentice Hall, 2009.
- ❑ G. Harrison, Next Generation Databases: NoSQL, NewSQL, and Big Data, Apress, 2016

# NoSQL Stores

- **Key-value** store – a set key-value pairs with no schema or exposed nesting
- **Column-family** store – the data model is based on a sparsely populated table whose rows can contain arbitrary column
- **Document stores** - arbitrary nested values, extensible records (XML, JSON object), no top-down schema is being enforced
- **Graph databases** – contains nodes and relationships (named and directed and optionally contain properties)

# Column-family Store

## Cassandra



**Column-family** store – the data model is based on a sparsely populated table whose rows can contain arbitrary column

# Cassandra : History

- Facebook (2007-2008)
  - Avinash, former Dynamo engineer
  - Motivated by “Inbox Search”
- Google Code (2008-2009)
  - Dark times
- Apache (2009-2018+)
  - Digg, Twitter, Rackspace, Others
  - Rapidly growing community
  - Fast-paced development
- Datastax (2011-2018+)
- <http://www.datastax.com>

# Cassandra: Combination


- Google BigTable's data model
  - Column Families – now “Tables” with scheme
  - Memtables - data in memory
  - SSTables – data in disks
- Amazon Dynamo
  - Distributed hashing table
  - Eventual Consistency model - NWR notation Can be tuned per-operation
  - Availability by Data Partitioning and elastic scalability
  - Replication and availability
    - Resilient in case of network disruption or node failure

# Cassandra - Main Charts

- Column data model
  - Data in tables, as in RDBMS,
  - Inability to combine relationships,
  - Poor consistency (lost updates, last write wins).
- Use
  - Optimized for recording,
  - It handles TimeSeries workload particularly well.
- API
  - CQL (a subset of SQL).
- Replication and availability
  - Resilient in case of network disruption or node failure
  - Distributed hashing table

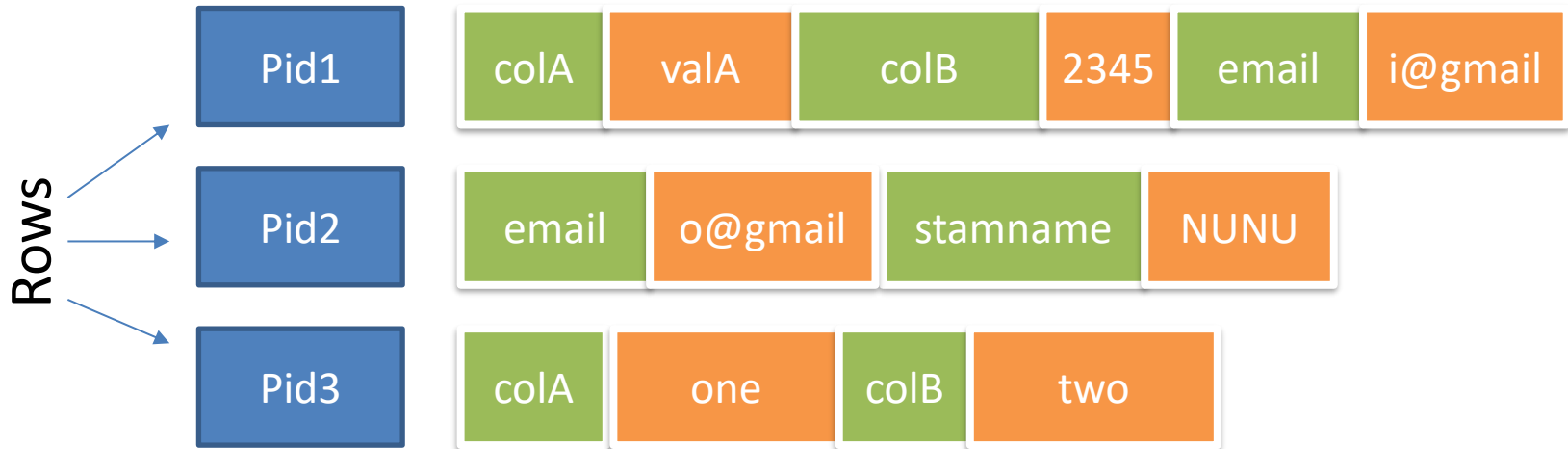
# Cassandra Data Model

**Cluster**: the set of machines cooperate on Cassandra

- **Keyspace** (“Namespace”)  **schema**
  - Table (“ColumnFamily”)
    - Row (indexed)
      - Key (row key)
      - Columns
        - Name (sorted)
        - Value,
        - TTL
        - timestamp



# Column Family(CF): Schema-free Sparse-table



- Flexible column naming
- Columns are always sorted by their name.
- Not required to have a specific column just because another row does
- Each column family is partitioned vertically among the nodes using its row key (partition key).

# Cassandra Data Model (1)

- **Partition key** refers to a single row on a single node.
- Column: name + value (+ timestamp).
- Variable number of columns.

Name (row key)		Columns	
Maya	Company Name	Street	
	Salesforce	King Shlomo	
Gil	Company Name	Street	
	Amazon	Yehuda Maccabi	
Emilie	Company Name	Street	Cel. Phone
	Salesforce	Miriam Profit	123
Tomer	Company Name	Street	Cel. Phone
	Salesforce	Miriam Profit	456

Scheme for CQL

Name (row key)	Company Name	Street	Cel. Phone
Maya	Salesforce	King Shlomo	Null
Gil	Amazon	Yehuda Maccabi	Null
Emilie	Salesforce	Miriam Profit	123
Tomer	Salesforce	Miriam Profit	456

# Cassandra Data Model (2)

- “update” operations: last-write-wins
  - R. H. Thomas. “A majority consensus approach to concurrency control for multiple copy databases”. In: *ACM Transactions on Database Systems* 4.2 (June 1979), pp. 180–209

Update 1 at timestamp 25

Name (row key)	Columns	
Emilie	“Company Name”	“Street”
	Salesforce	Miriam Profit
	(timestamp 25)	(timestamp 25)

Update 2 at timestamp 30

Name (row key)	Columns	
Emilie	“Cel Phone”	“Street”
	456	Bnei Dan
	(timestamp 30)	(timestamp 30)

Content in the database

Name (row key)	Columns		
Emilie	“Company Name”	“Street”	“Cel Phone”
	Salesforce	Bnei Dan	456
	(timestamp 25)	(timestamp 30)	(timestamp 30)

# API until 2011

- `get() : Column`
  - get the Col or SC at given ColPath  
COSC cosc = client.get(key, path, CL);
- `get_slice() :`  
`List<ColumnOrSuperColumn>`
  - Returns a slice (list of columns per key)
- `multiget_slice() : Map<key, List<CoSC>>`
  - get slices for *list of keys*, based on SlicePredicate
- `get_range_slices() : List<KeySlice>`
  - returns multiple Cols according to a *range*
  - range is startkey, endkey, starttoken, endtoken:

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

remove

# CQL - Cassandra

- Cassandra narrowed its openness and moved to schema forms.
- Cassandra Query Language (CQL)
  - CQL syntax is based on SQL – it eases customer's readability
  - Though the underline data model is completely different
  - ➔ Can't avoid getting into design details

# Cassandra data model (3)

```
CREATE KEYSPACE IF NOT EXISTS Test
```

```
    WITH REPLICATION = { 'class' : 'SimpleStrategy',  
                          'replication_factor' : 1 };
```

```
USE Test;
```

```
CREATE TABLE IF NOT EXISTS Users (
```

```
    companyName text,
```

```
    name text,
```

```
    phone int,
```

```
    street text,
```

```
    PRIMARY KEY (name)
```

```
);
```

```
CREATE TABLE IF NOT EXISTS Users (
```

```
    companyName text,
```

```
    name text PRIMARY KEY,
```

```
    phone int,
```

```
    street text,
```

```
);
```

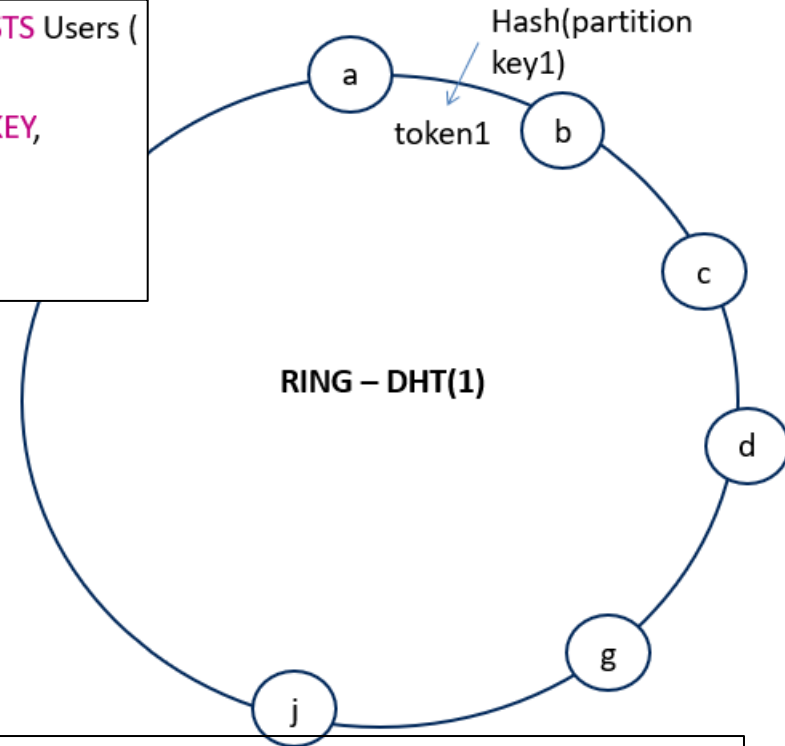
The first component of the primary key is the Partition key.

**Partition Key** Determines how data is distributed across the nodes in the cluster.

# Cluster Architecture: Data Partitioning

- Cassandra is a DHT system that distributes column family **rows** among a group of nodes and is organized by DHT table
- A primary key identifies rows. The primary key determines which node the data is stored on.

```
CREATE TABLE IF NOT EXISTS Users (  
  companyName text,  
  name text PRIMARY KEY,  
  phone int,  
  street text,  
);
```



## Consistent Hashing

- Hash partition keys into tokens using a **consistent hashing** function that provides new hash values uniformly
- **Murmur3** is a family of non-cryptographic, fast hash functions for distribution (so hash collisions are rare) and high-performance
- Cassandra uses this function with a key space of 64 bits, with the range of  $-2^{63}$  to  $2^{63}$

# Cluster: Consistent Hashing

For example, if you have the following data:

jim	age: 36	car: camaro	gender: M
carol	age: 37	car: bmw	gender: F
johnny	age: 12	gender: M	
suzy	age: 10	gender: F	

Cassandra assigns a hash value to each partition key:

Partition key	Murmur3 hash value
jim	-2245462676723223822
carol	7723358927203680754
johnny	-6723372854036780875
suzy	1168604627387940318



# Cluster: Consistent Hashing

Each node in the cluster is responsible for a range of data based on the hash value:

Node	Murmur3 start range	Murmur3 end range
A	-9223372036854775808	-4611686018427387903
B	-4611686018427387904	-1

Node	Murmur3 start range	Murmur3 end range
C	0	4611686018427387903
D	4611686018427387904	9223372036854775807

# Cassandra Installation

Install and Run Cassandra from GitHub

Log in to your GitHub account and search for the <https://github.com/mirslivjce/TeachCassandraClusterWDocker> project.

# Class Ex1 (1)

```
CREATE KEYSPACE IF NOT EXISTS test1
    WITH REPLICATION = { 'class' : 'SimpleStrategy',
                        'replication_factor' : 1 };

USE Test;

CREATE TABLE IF NOT EXISTS Users (
    companyName text,
    name text,
    phone int,
    street text,
    PRIMARY KEY (name)
);
```

# Class Ex1 (2)

```
INSERT INTO Users (companyName, name, street)
```

```
VALUES ('Amazom', Gil, 'A St');
```

```
INSERT INTO Users (companyName, name, phone, street)
```

```
VALUES ('Amazom', 'Maya', 609, 'A St');
```

```
INSERT INTO Users (companyName, name, street)
```

```
VALUES ('Salesforce', 'Emilie', 'B St');
```

```
INSERT INTO Users (companyName, name, phone, street)
```

```
VALUES ('Amazom', 'Tomer', 720, 'C St');
```

```
SELECT * FROM Users;
```

```
SELECT * FROM Users WHERE name='Tomer';
```

```
SELECT * FROM Users WHERE companyName='Amazon';
```




# Class Ex1 (3) – Partition Key and Clustering Key

```
DROP TABLE Users;
```

```
CREATE TABLE IF NOT EXISTS Users (  
    companyName text,  
    name text,  
    phone int,  
    street text,  
    PRIMARY KEY (companyName, name)  
);
```

```
INSERT INTO Users ...;
```



PRIMARY KEY (companyName, name)  
companyName – partition key  
name – clustering column

# Class Ex1 (4)

SELECT \* FROM Users;

SELECT \* FROM Users  
WHERE name='Tomer';



SELECT \* FROM Users  
WHERE companyName='Amazon';



SELECT \* FROM Users  
WHERE companyName='Amazon' AND name='Tomer';



SELECT \* FROM Users  
WHERE companyName='Amazon' AND Phone=72;



# Class Ex1 (5)

**PRIMARY KEY** ((pk1, pk2, ...), cc1, cc2, ...)

- (pk1, pk2, ...) – partition key
- cc1, cc2, ... – clustering column

Note the difference:

**PRIMARY KEY** (companyName, name, Phone)

**PRIMARY KEY** ((companyName, name), Phone)



Tuple partition key

# Class Ex1 (6) – Index is not clustering

```
DROP TABLE Users;
```

```
CREATE TABLE IF NOT EXISTS Users (  
    companyName text,  
    name text,  
    phone int,  
    street text,  
    PRIMARY KEY (companyName, name)  
);
```

```
SELECT * FROM Users WHERE Phone=720;
```



```
CREATE INDEX ON Users (phone);
```

→

```
SELECT * FROM Users WHERE Phone=720;
```



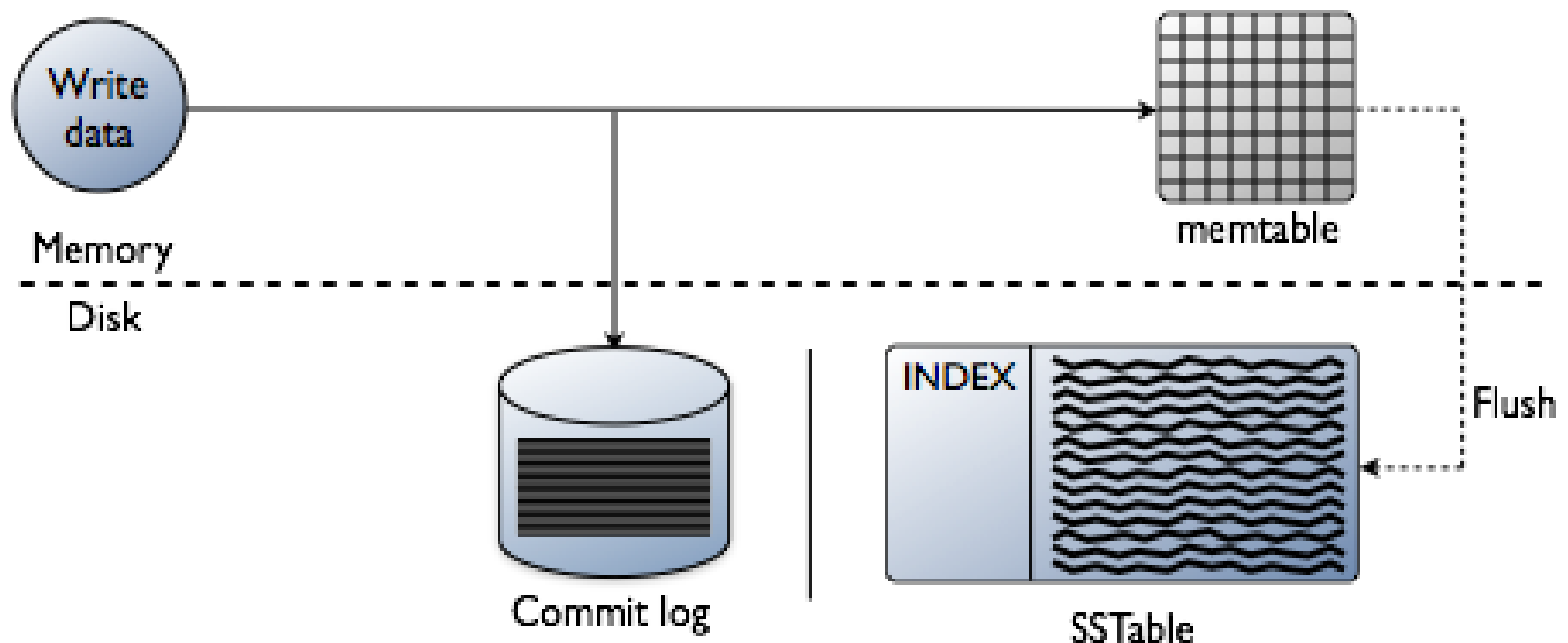


# Class Ex1 (7) – Differences between clustering and index

Clustering Key	Index (Secondary Index)
Sorts rows inside a partition	Allows lookup by non-primary key columns
Only inside a single partition	Global (across partitions)
Very fast, efficient scanning	Beware – Can be very slow
Natural for range queries inside one partition	Used when you don't know the partition key in advance
Part of the primary storage structure	Require additional external structure or management

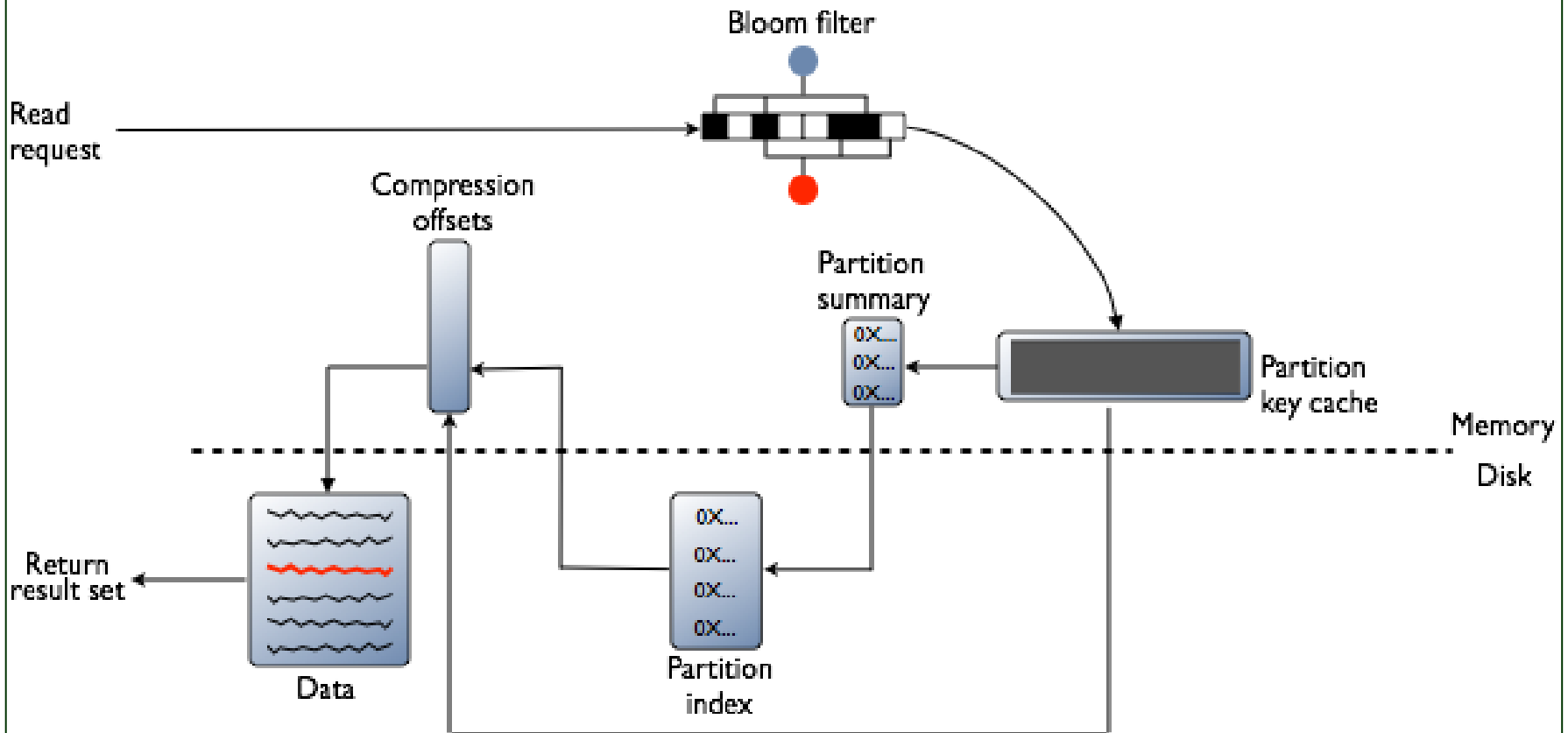
# Cassandra Node – Writes(1)

- **Writes are FAST!!!**
- Data is written to memory (RAM) - Memtable
- Upon threshold, the memtable data, which includes indexes, is flushed to disk.



# Cassandra Node – Reads (3)

- **Read are Efficient!!!**
- Using sophisticated caches and filters in memory (RAM)



# Query-Oriented Design

- Column stores are query-oriented
  - Start from your query and work backwards
  - At scale, everything depends on indexes and de-normalization
  - **De-normalization** means duplicating the data among several nodes in order to answer the query immediately.

# Refs

- [https://docs.datastax.com/en/cql-oss/3.1/cql/cql\\_reference/cqlshCommandsTOC.html](https://docs.datastax.com/en/cql-oss/3.1/cql/cql_reference/cqlshCommandsTOC.html)
- <https://cassandra.apache.org/doc/stable/cassandra/tools/cqlsh.html>
- <https://cassandra.apache.org/doc/stable/cassandra/cql/index.html>