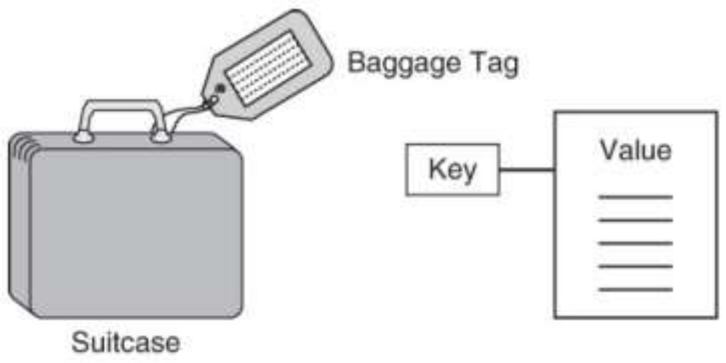
NoSQL Data Stores

LECTURE 2

I - Key-Value Data Stores

What Is a Key-Value Store

- Key-value stores are the *simplest* form of NoSQL databases.
- Used when *all access* to the database is *via a key*.
- Modelled on two components:
  - Key: short byte sequence works as a reference to a value.
  - Value: byte sequence that has been associated with a key.

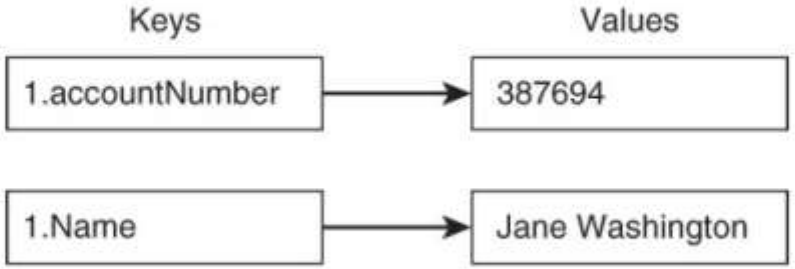


- You can either *put* a value for the key, *get* a value of a key, or *delete* a key from the data store.
- The *value is a blob* that stores data, without caring or knowing what's inside
- It's the *responsibility of the application* to understand what was stored in the value.
- Generally, Key-value stores have *great performance* and can be easily *scaled*.

Trade simplicity for speed and scale

Key

- Keys are *identifiers* associated with values.
- Some key-value databases support buckets, or collections, for creating separate *namespaces* within a database.
- Keys must be *unique* within a namespace



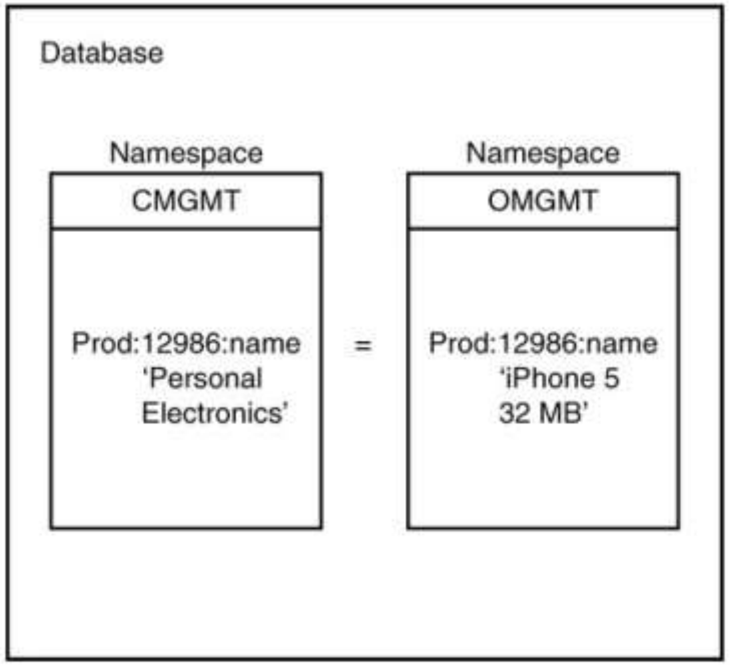
Value

- A value is an *object*, a set of bytes, that has been associated with a key.
- Values can be integers, floating-point numbers, strings of characters, binary large objects (BLOBs), semi-structured constructs such as JSON objects, images, audio, and just about any other data type you can represent as a series of bytes.
- There *is no enforcement* on the *data types* of values.
- Key-value databases typically do not enforce checks on data types of values.

Namespace

- A namespace is a *collection of key-value* pairs that has no duplicate keys but duplicate values in a namespace are permissible.
- Namespaces are helpful when multiple applications use a key-value database.
- Namespaces implicitly defining an additional *prefix* for keys the indicated the application.

custMgmt: Prod:12986:name
ordMgmt: Prod:12986:name



- Here custMgmt and ordMgmt are two namespaces.

How to Construct a key

- A key must be *Immutable* and *Unique* within a namespace.
- Unlike primary keys in relational DBs where it is not preferred to have a meaningless key.

```
Cart[12387] = 'SKU AK8912j4'
```

- It is preferable to use *meaningful keys* that entail information about attributes.

```
Cust:12387:firstName
```

- Keys Should Follow a *Naming Convention*, suggested formula:

```
Entity Name + ':' + Entity Identifier +':' + Entity Attribute
```

  - The delimiter : does not have to be a ':' but it is a common practice.

How to Construct a key

- Avoid keys that are too short or too long.
  - *Short keys* are more likely to lead to *conflicts* in key names.
  - *Long keys* will use *more memory* and key-value databases tend to be memory-intensive systems already.
- Some key-value databases *restrict the size* of keys:
  - FoundationDB limits the size of keys to 10,000 bytes.
  - In Redis, the maximum allowed key size is 512 MB.
- Others *restrict the data types* that can be used as keys:
  - Riak treats keys as binary values or strings.
- Capturing the right data in your key-value pairs is important both for meeting functional requirements and for ensuring adequate performance of your application.

Designing Structured Values

- Key-value databases do not expect you to specify types for the values you store.

```
'1232 NE River Ave, St. Louis, MO'                                   ← String
('1232 NE River Ave', 'St. Louis', 'MO')                             ← List
{ 'Street:' : '1232 NE River Ave', 'City' : 'St. Louis','State' : 'MO' }   ← JavaScript Object
```

- Different implementations of key-value databases have different restrictions on values.
- You should consider the workload on your server as well as on developers when designing applications that use key-value data stores.

Designing Structured Values - Example

- Consider an application development project in which the customer address is needed about 80% of the time when the customer's name is needed.
- You can set the value with several different keys and to have a function that retrieves both the name and the address in one function call.

```
cust: 198277:name =  'Jane Anderson'
cust: 198277:Addr   = '39 NE River St. Portland, OR 97222'
```

- OR, you can store a customer name and address together in one key.

```
cust: 198277:name  =  'Jane Anderson'
cust: 198277:nameAddr = { 'Jane Anderson' , '39 NE River St. Portland, OR 97222'}
```

- This is a more complex value structure than using several different keys but has significant advantages in some cases. By storing a customer name and address together, you might reduce the number of disk seeks that must be performed to read all the needed data.

Limitations on Searching for Values

What if you don't know the key?

- Operations on values are all based on keys:
  - *Get* the value for a key
  - *Put* a value for a key
  - *Delete* a key from the data store.
- Key-value data stores do not support query languages for searching over values.
- If you want to do more, such as searching inside the value, you have only two solutions:
  - Implementing the required search operations in your application – *Inefficient for large ranges of data.*
  - Having a built-in search system that uses an index for rapid retrieval – *Not in all Key-value DBs*

Example

- Assuming that you set a customer address as a string:

cust:9877:address = ‘1232 NE River Ave, St. Louis, MO’

- If you want to do more, such as search for an address in which the city is “St. Louis,”

```
define findCustomerWithCity(p_startID, p_endID, p_City):
for id in p_startID to p_endID:
     address = appData[‘cust:’ + id + ‘:address’];
     if inString(p_City, Address):
        addToList(Address,returnList );
# after checking all addresses in the ranges specified
# by the start and end ID return the list of addresses
# with the specified city name.
return(returnList);
```

A pseudocode function for searching for customers in a particular city

Word	Keys
‘IL’	‘cust:2149:state’, ‘cust:4111:state’
‘OR’	‘cust:9134:state’
‘MA’	‘cust:7714:state’, ‘cust:3412:state’
‘Boston’	‘cust:1839:address’
‘St. Louis’	‘cust:9877:address’, ‘cust:1171:address’

A search index helps efficiently retrieve data when selecting by criteria based on values.

Dealing with Ranges of Values

- Dealing with ranges of values is another thing which should be considered as most key-value databases do not support range queries.
- Consider using values that indicate ranges when you want to retrieve groups of values.
- For example, if you want to retrieve all customers who made a purchase on a particular date.
- You might want to include a six-digit date in a key:

```
Cust151120:9877 = …
```

- Would be better than:

```
cust:9877:date = ‘151123’
```

Why??

Values With Big Aggregates

- Using structured data types, such as lists and sets, can improve the overall efficiency of some applications by minimizing the time required to retrieve data.
- Large values can Lead to inefficient read and write operations.
- Consider a data structure that maintains customer order information in a single value, such as the following:

```
{
        “Order number" : 1,
        “Order details" : [
                {"Item name" : "lightsaber", "Item quantity" : 1, "Item price" : 100},
                {"Item name" : "black cloak", "Item quantity" : 2, "Item price" : 50},
                {"Item name" : "air filter", "Item quantity" : 10, "Item price" : 2}
        ],
        "Customer details" : {
                "Customer name" : "Dart Vader"
                "Customer location" : "Star Destroyer"
        }
}
```

Data is read in blocks.
Blocks may store a large number of small-sized values or few large-sized values.
The former can lead to better performance if frequently used attributes are available in the cache.

If, however, you load a large value into the cache and only reference a small percentage of the data, you are essentially wasting valuable memory.

Limitations of Key-Value Databases

- Key-value databases are the simplest of the NoSQL databases. However, it also brings with them important limitations:
  - The only way to look up values is by key.
  - Some key-value databases do not support range queries.
  - There is no standard query language comparable to SQL for relational databases.
- Different key-value database vendors and open-source project developers take it upon themselves to devise ways to mitigate the disadvantages of these limitations.

Popular Key-Value Stores

- Riak - Distribution
- Redis – Data Structure server
- Memcached DB - In-memory key-value store
- Berkeley DB – Oracle
- Aerospike – fast key-value for SSD disks
- LevelDB – Google key-value store
- DynamoDB – Amazon key-value store
- VoltDB – Open Source Amazon replica

DB-Engines Ranking of Key-value Stores

https://db-engines.com/en/ranking/key-value+store/all

Redis - *data Structures Server*

- Redis is an open source, BSD licensed, advanced key-value cache and store. It is often referred to as a *data structure server* since keys can contain strings, hashes, lists, sets, sorted sets, bitmaps and hyperloglogs.
  - Written C++ with BSD License
  - Keys can contain strings, hashes, lists, sets, sorted sets, bitmaps and hyperloglogs.
  - It works with an *in-memory*.
  - Data *can be persisted* either by dumping the dataset to disk every once in a while, or by appending each command to a log.
  - Created by Salvatore Sanfilippo (Pivotal)

Who Uses Redis in Production

twitter

trivago

Udemy

snapchat

WordPress

Azure Redis Cache

stackoverflow

Uber

Airbnb

Instagram

Pinterest

GitHub

II-Document Databases

Document Databases - Document-Oriented Databases

- Document databases is a *flexible* form of NoSQL datastores that can manage *more complex data structures* than key-value databases.
- You *do not have to define a fixed schema* before you add data to the database.
- Document databases *use a key-value approach* for storing data but with important differences: document database stores values as documents that are *examinable*.
- Some similar features to relational databases
  - E.g. It is *possible to query* and filter collections of documents much as you would do with tuples in a relational table.

What is a Document?

- A document is a set of *ordered key-value pairs*.
- documents has a *standard format* such as Extensible Markup Language (XML), JavaScript Object Notation (JSON), or Binary representation of JSON (BSON).
- Documents may be *basic data types* (such as numbers, strings, and Booleans) or *structures* (such as arrays and objects).
- The order of key-value pairs matters in determining the identity of a document.

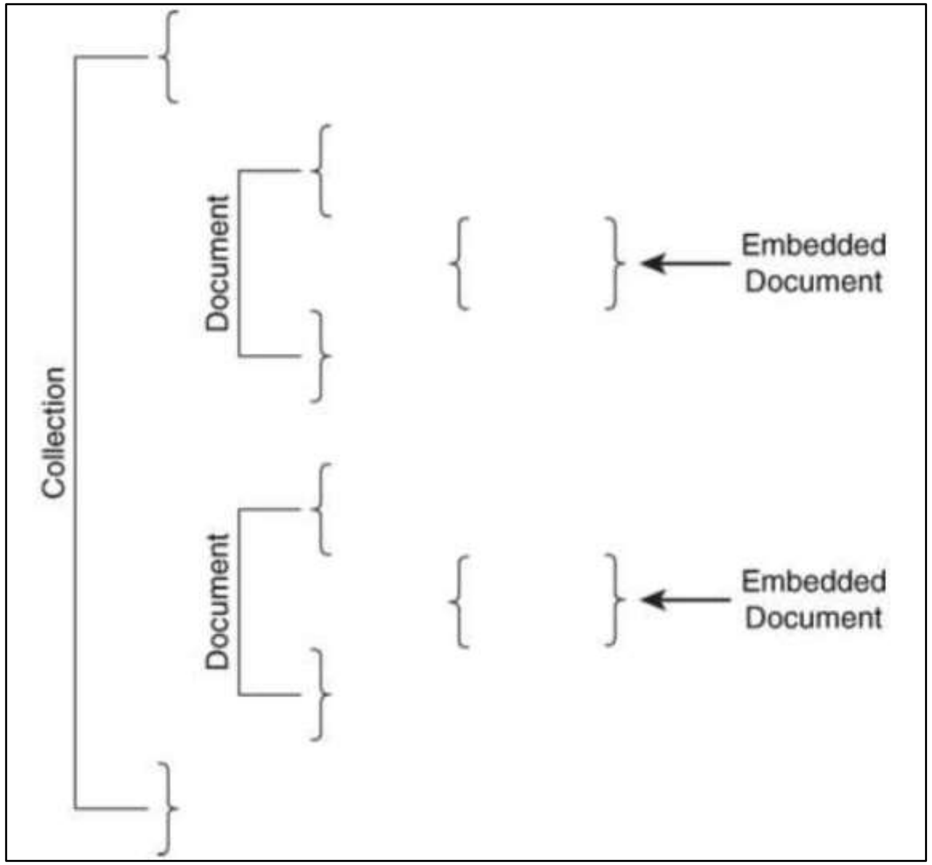
What is a Document?

- Consider a simple example of a customer record that tracks the customer ID, name, address, first order date, last order date, and customer Tel. numbers. Using JavaScript Object Notation (JSON)
  - Documents consist of name-value pairs separated by commas.
  - Documents start with a { and end with a }.
  - Names are strings, such as "customer_id" and "address".
  - Values can be numbers, strings, Booleans (true or false), arrays, objects, or the NULL value.
  - The values of arrays are listed within square brackets [and].
  - The values of objects are listed as key-value pairs within curly brackets, that is, { and }.

```
{
“customer_id”:187693,
“name”: “Kiera Brown”,
“address” :
        {
        “street” : “12 Sandy”,
        “city” : “Vancouver”,
        “zip” : “99121”
        },
“first_order” : “01/15/2013”,
“last_order” : ” 06/27/2014”,
“Tel.No" : [“521691", “702241",
         “123456"]
}
```

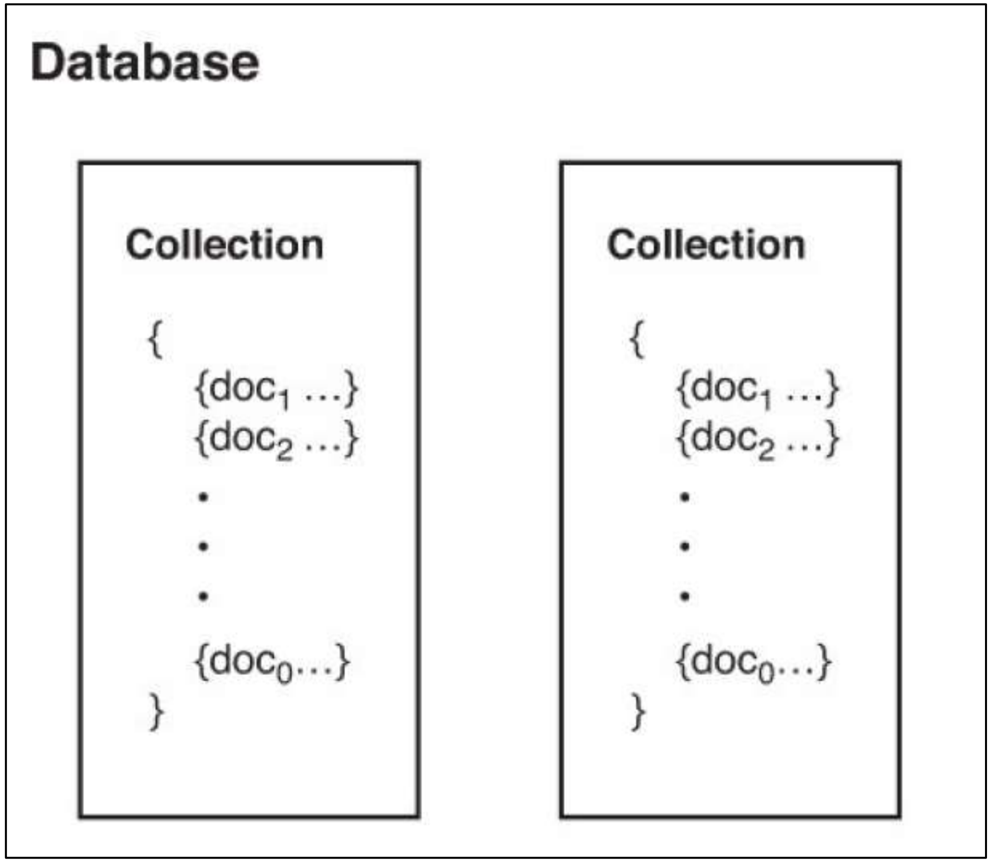
Embedded Document

- Embedded documents are documents within a document.
- An embedded document enables document database users to store *related data in a single document*.
- This allows the document database to *avoid the JOIN operation*.
- They are used to *improve database performance* by storing together data that is frequently used together.



Collections

- A collection is a *group of documents*.
- The *database is the container for collections* and collections are containers for documents.
- Collections allow you to *easily operate* on groups of documents.
- Collections support additional data structures that make such operations more efficient.
  - For example, a more efficient approach to scanning all documents in a collection is to use an index.



Collections

- The documents within a collection are usually related to the *same subject entity*, such as employees, products, logged events, or customer profiles.
- documents in the same collection *do not need to have identical structures*, but they should share some common structures.
- It is possible to store *unrelated* documents in a collection, but this is not advised.

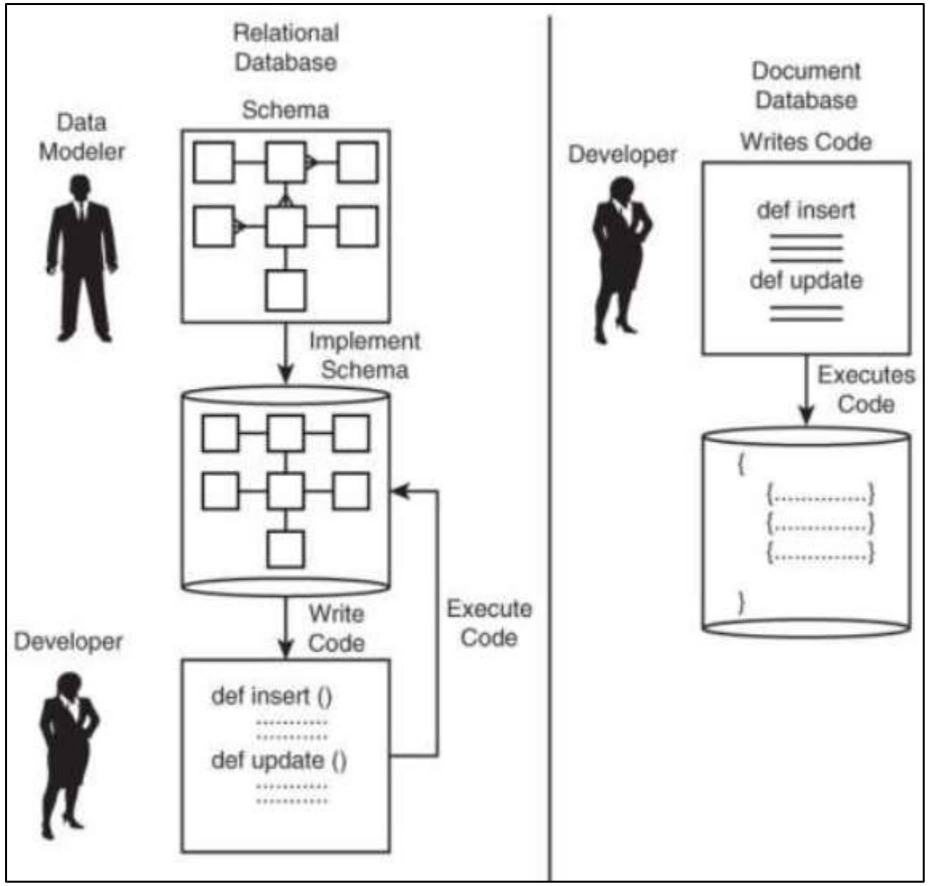
```
{
"customer_id":187694,
"name": "Bob Brown",
"address" : {
        "street" : "1232 Sandy Blvd.",
        "city" : "Vancouver",
        "zip" : "99121"
        },
"first_order" : "02/25/2013",
"last_order" : " 05/12/2014"
}

{
"customer_id":179336,
"name": "Hui Li",
"address" : {
        "street" : "4904 Main St.",
        "city" : "St Louis",
        "zip" : "99121"
        },
"first_order" : "05/29/2012",
"last_order" : " 08/31/2014",
"loyalty_level" : "Gold",
"discount" : 250.00
}
```

Schemaless / Polymorphic Schema

- Document databases do not require a formal schema definition step – *Schemaless*.
- Instead, developers can create collections and documents in collections by simply inserting them into the database.

Schemaless means more flexibility and more responsibility

- Document database supports multiple types of documents in a single collection - *Polymorphic schema*.



Managing Multiple Documents in Collections

- One of the essential parts of modelling document databases is *deciding how you will organize your documents into collections*.
- Document database designers optimize document databases to *quickly* add, remove, update, and search for documents.
- They are also designed *for scalability*, so as your document collection grows, you can add more servers to your cluster to keep up with demands for your database.
- The *full potential* of document databases becomes apparent when you work *with large numbers of documents*
- Poor collection design can adversely affect performance and slow your application.

III- Column Family Databases

Google BigTable

- Companies such as Google, Facebook, Amazon, and Yahoo! must contend with demands *for Very Large DataBase (VLDB) management solutions*.
- In 2006, Google published a paper entitled “BigTable: A Distributed Storage System for Structured Data:
    - http://research.google.com/archive/bigtable.html.
- Google designed this database for several of its large services, including web *indexing*, *Google Earth*, and *Google Finance*.
- *BigTable* became the model for implementing very large-scale NoSQL databases.
- Cassandra and Apache HBase are the most common NoSQL Wide-Column stores that are descending from BigTable.

Column Family Databases

- Two reasons make Column Family the *most confusing type of databases* that we will deal with:

  1. Column Family databases *share a lot of terminologies with Relational Databases*, but they are very different.

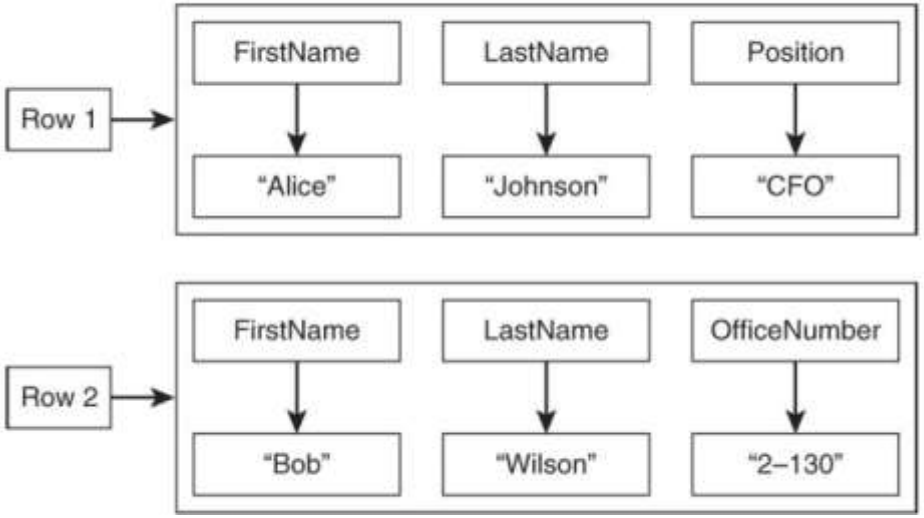
  2. There is a RDBMS technology known as *“Column-Store”* that people frequently conflict with “Column-Family”.

- People use terms like *Columnar*, *Column-Oriented*, and *Column Store* pretty much interchangeably to describe these two very different types of databases.

Columns and Column Families

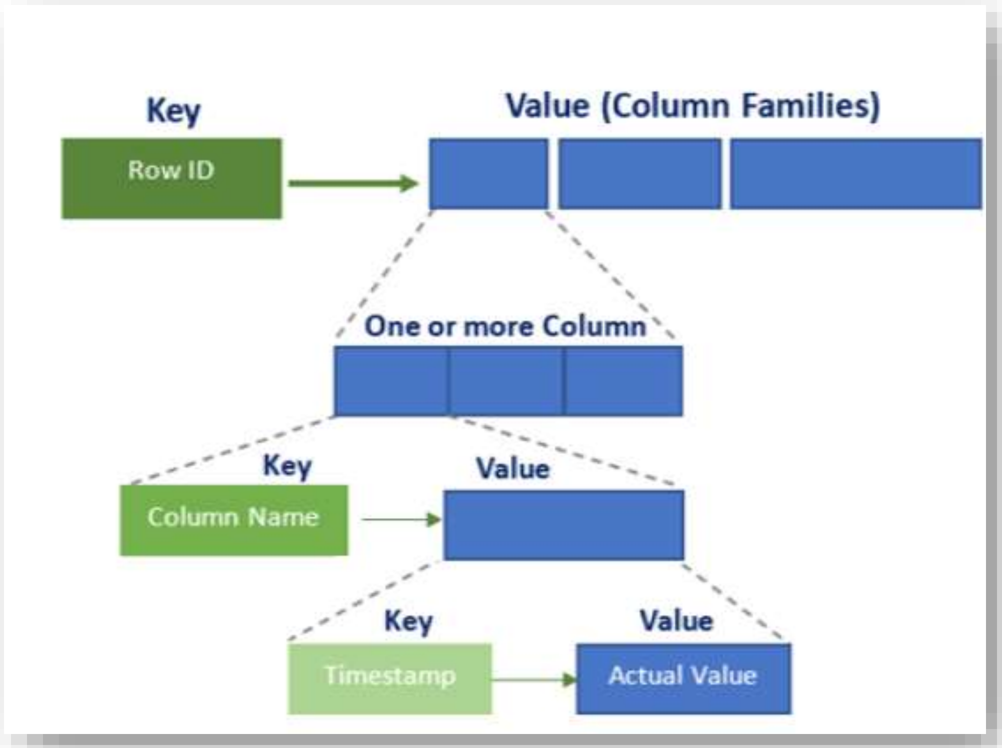
- The *most complex type of the NoSQL databases* in terms of the basic building block *structures*.
- Column family databases share some terms with relational databases, such as rows and columns.

- A *column is a basic unit of storage* in a column family database.
- A *set of columns* makes up a *row*. Rows can have the *same* columns, or they can have *different* columns.
- When there are *large numbers of columns*, it can help to group them into collections of related columns called *Column families*.
- Column family databases *do not require* a predefined *fixed schema*.

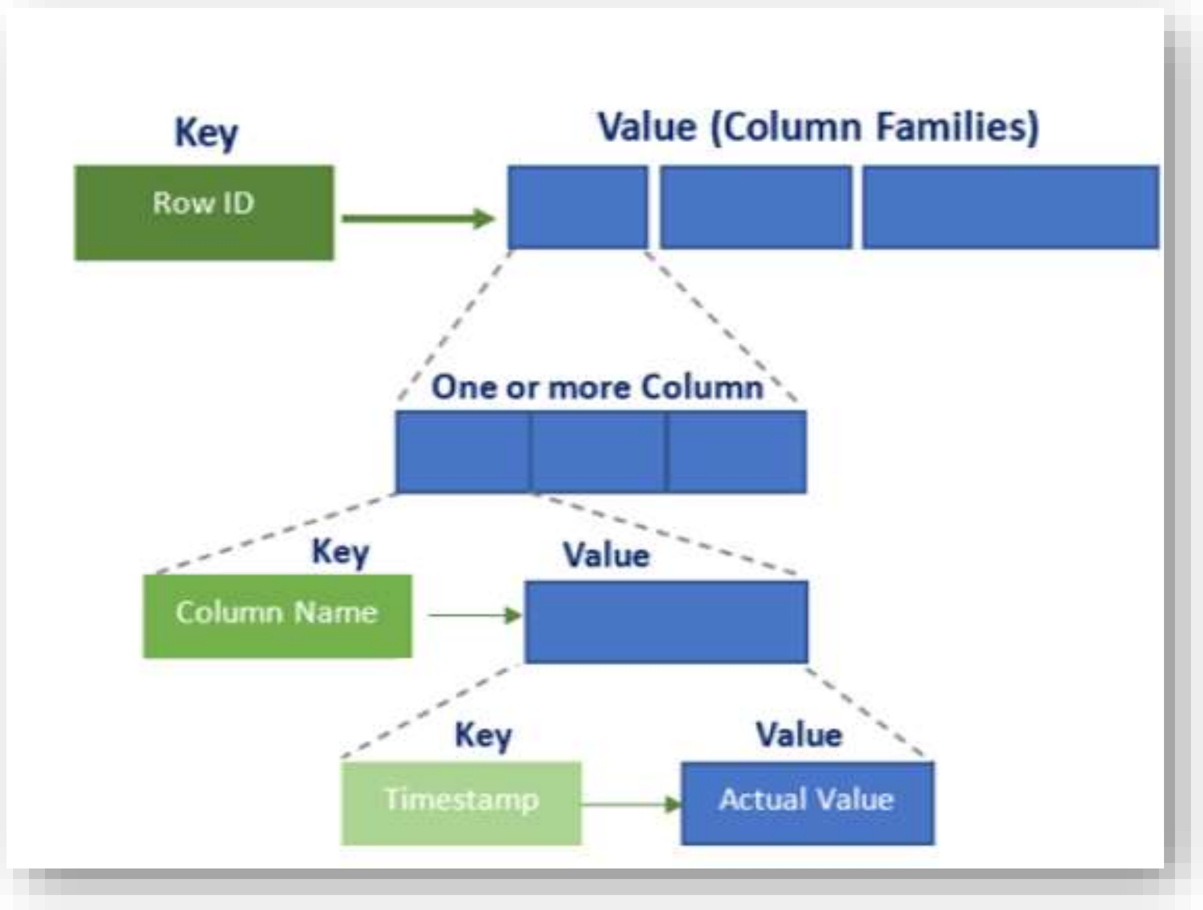


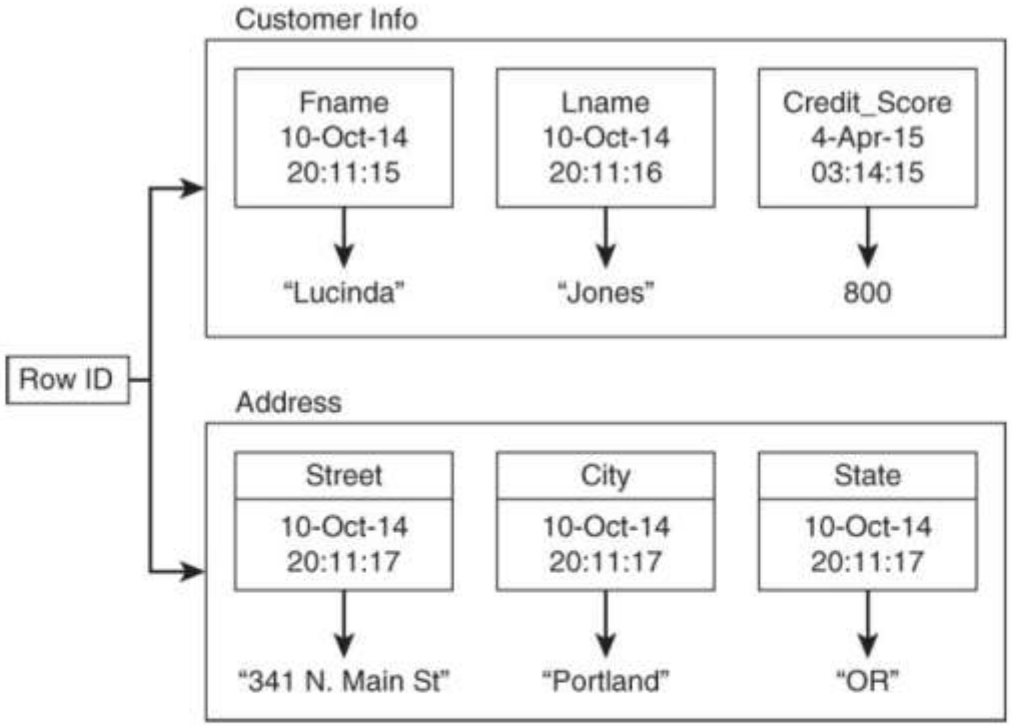
Data Model

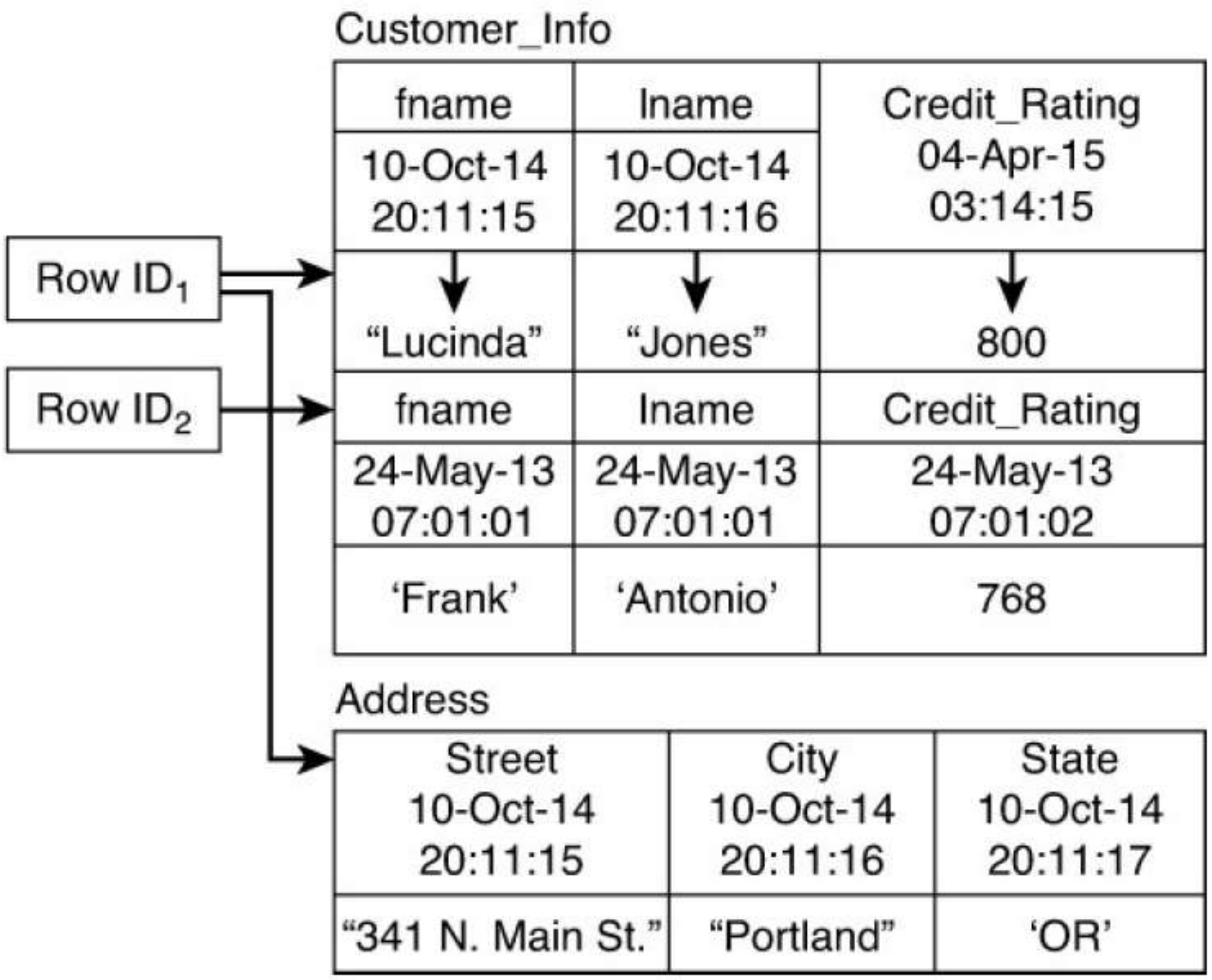
- A *row* is a Key-value pair where the key is the primary identifier of the recor, while the value is one or more column families.
  - *Each column* family is a *collection* of one or more *columns*.
    - *Each column*, on the other hand, is a key-value pair. A key represents the column name, and the value is another key-value pair.
      - The key is a timestamp that represents the version of this column and the value is the actual data. In this case., it will be the ID, the name, or the Age from the table.



(TABLE, ROWKEY, [COLUMN FAMILY], COLUMN, TIMESTAMP) → VALUE


Key
Row ID
Value (Column Families)
One or more Column
Key
Column Name
Value
Key
Timestamp
Value
Actual Value


Customer Info
Fname
10-Oct-14
20:11:15
"Lucinda"
Lname
10-Oct-14
20:11:16
"Jones"
Credit_Score
4-Apr-15
03:14:15
800
Row ID
Address
Street
10-Oct-14
20:11:17
"341 N. Main St"
City
10-Oct-14
20:11:17
"Portland"
State
10-Oct-14
20:11:17
"OR"


Customer_Info
fname
10-Oct-14 20:11:15
lname
10-Oct-14 20:11:16
Credit_Rating
04-Apr-15 03:14:15
"Lucinda"
"Jones"
800
Row ID1
Row ID2
fname
24-May-13 07:01:01
lname
24-May-13 07:01:01
Credit_Rating
24-May-13 07:01:02
'Frank'
'Antonio'
768
Address
Street
10-Oct-14 20:11:15
City
10-Oct-14 20:11:16
State
10-Oct-14 20:11:17
"341 N. Main St."
"Portland"
'OR'

Basic Components of Column Family Databases

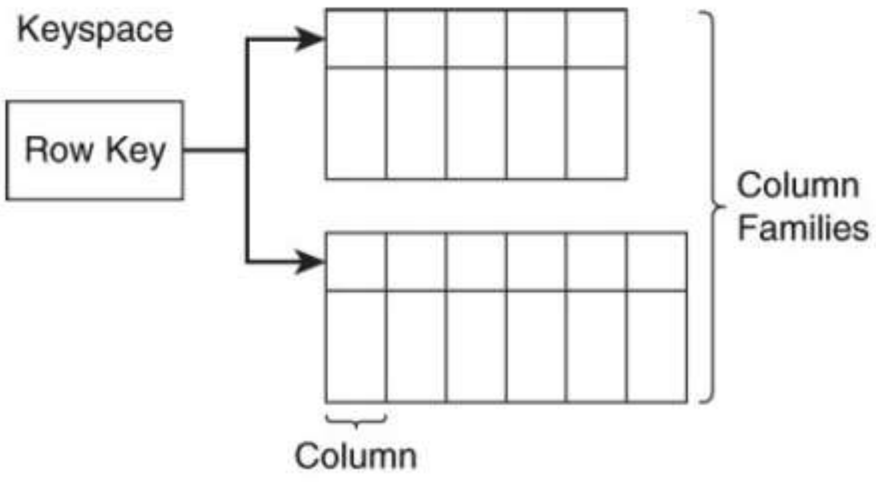
- The basic components of a column family database are the data structures developers deal with the most:

Keyspace

- A keyspace is a *top-level container* that logically holds column families, row keys, and related data structures. Typically, there is *one keyspace per application*.
- It is analogous to a *schema* in a relational database.

Row Key

- A row key *uniquely identifies a row* and has a role in determining the *order* in which data is stored.
- Row keys are also used to *partition* and order data.



Basic Components of Column Family Databases

Column

- A column is the data structure for storing a single value in a database. Columns have three parts:
  - **Column name**: serves the same function as a key in a key-value pair: It refers to a value.
  - **Timestamp** or other version stamp: represents the version of the column and it is a way to order values of a column
  - **Value**: which is the actual value.

Column Families

- Column families are *collections of related columns*. Columns that are frequently used together should be grouped into the same column family.

Differences Between Column Family and Relational Databases

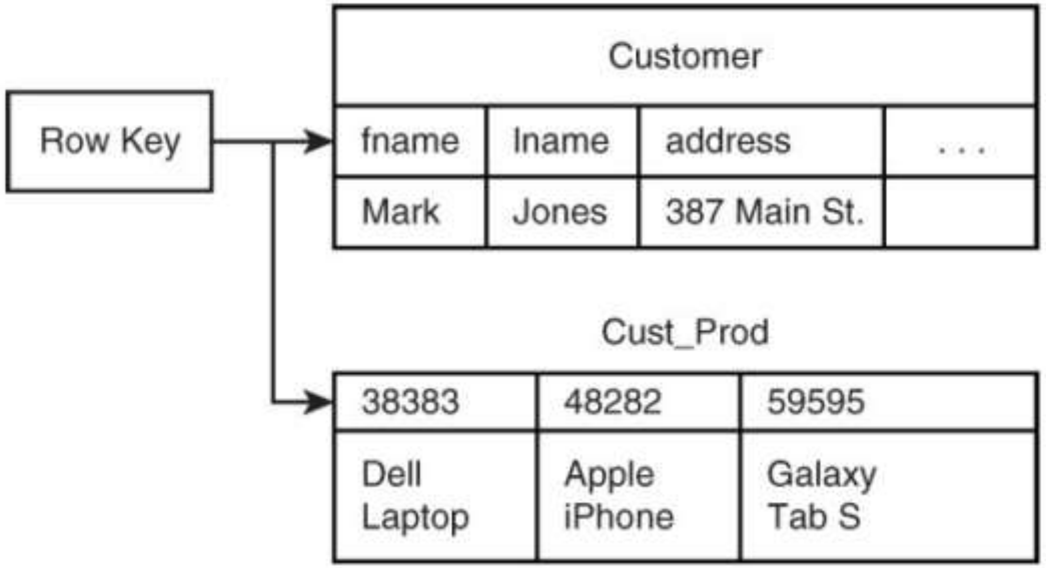
Relational Databases	Column Family Databases
Tables in relational databases have a relatively *fixed structure*.	the set of columns in a column family table can *vary* from one row to another.
In relational databases, data about an object can be stored in multiple tables due to *Normalization*.	Column family databases are typically *denormalized* or structured so that all relevant information about an object is in a single, possibly very wide, row.
CCRUD operations are through *SQL*	Query languages for column family databases may look similar to SQL.

Designing for Column Family Databases

- Column family databases are implemented differently than relational databases.
- *Users* are the ones who *drive the design* of a column family database.
  - Column family databases are implemented as *sparse*, *multidimensional maps*.
  - Columns can *vary* between rows and they cab be *added dynamically*.
  - Data values are *indexed* by *row identifier*, *column name*, and *time stamp*.
  - Joins are not used; data is *denormalized* instead.

Denormalize Instead of Join

- Tables model entities, so it is reasonable to expect to have *one table per entity*.
- Column family databases often need *fewer tables* than their relational counterparts.
- This is because column family databases *denormalize* data to avoid the need for joins.
- For example, in a column family database, *many-to-many* relationships are captured by denormalizing data.



Keep an Appropriate Number of Column Value Versions

- Column values are *timestamped* so you can determine the *latest and earliest values*.
- This feature is useful if you need to *roll back changes* you have made to column values.
- You should keep as many versions as your application requirements dictate, but no more. Additional versions will obviously require more storage.
- In some column databases, such as Hbase, the number of versions maintained is controlled by database parameters, which can be changed according to application requirements.

Column Family		
Column $Name_1$	Column $Name_2$	Column $Name_3$
$value_{1a}$: $timestamp_{1a}$	$value_{2a}$: $timestamp_{2a}$	$value_{3a}$: $timestamp_{3a}$
$value_{1b}$: $timestamp_{1b}$	$value_{2b}$: $timestamp_{2b}$	$value_{3b}$: $timestamp_{3b}$
$value_{1c}$: $timestamp_{1c}$	$value_{2c}$: $timestamp_{2c}$	$value_{3c}$: $timestamp_{3c}$

Avoid Complex Data Structures in Column Values

- Unlike document databases where *embedded documents* are commonly used with documents, it is not recommended to store this type of structure in a column value unless there is a specific reason to maintain this structure.
- If you expect to query or *operate on the values within the structure*, then it is better to decompose the structure.
- Using separate columns for each attribute makes it easier to create *secondary indexes* on each of those values.
- Also, separating attributes into individual columns allows you to *use it in different column families* if needed.

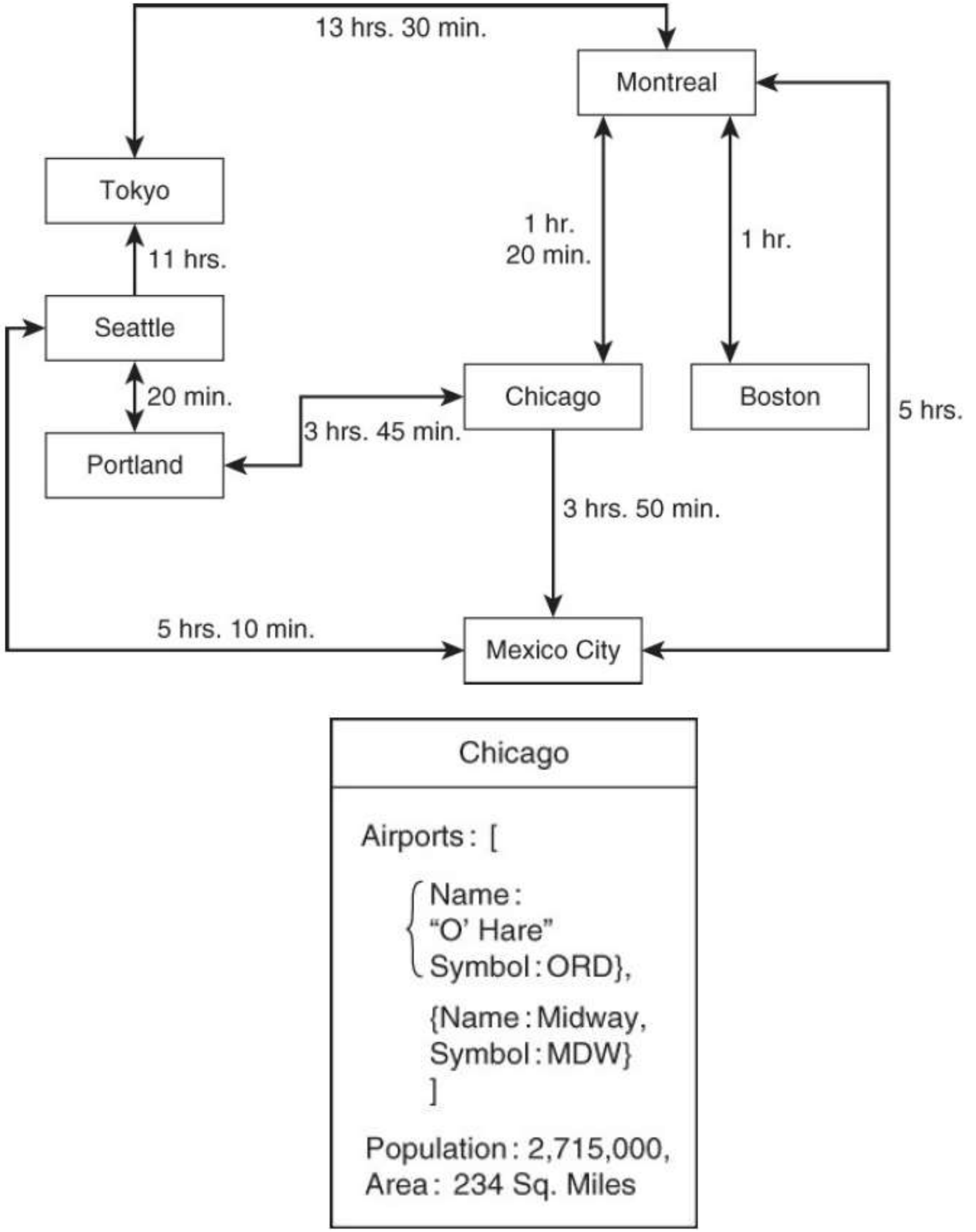
IV - Graph Databases

Graph Databases

- A graph database is a *specialized type* of database that is based on the *Graph Theory*.
- The two basic building blocks of graphs are vertices and edges.
- Even with such a *simple model*, graphs are suitable for modelling several domain areas. This forms the foundation for their usefulness as a *major type of NoSQL database*.
- Graph databases can be used for both *OLAP* (since are naturally multidimensional structures) and *OLTP*.
- Systems tailored to OLTP (e.g., Neo4j) are generally optimized for transactional performance, and tend to guarantee *ACID properties*.

Graph DB

- Graph database uses topological structures called *vertices and edges* (nodes and relationships).
  - A *Vertex* is an object that has an identifier and a set of attributes.
  - An *Edge* is a link between two vertices and contains properties about that relationship.
- Both *nodes* and *relationships* can have *complex structures*.

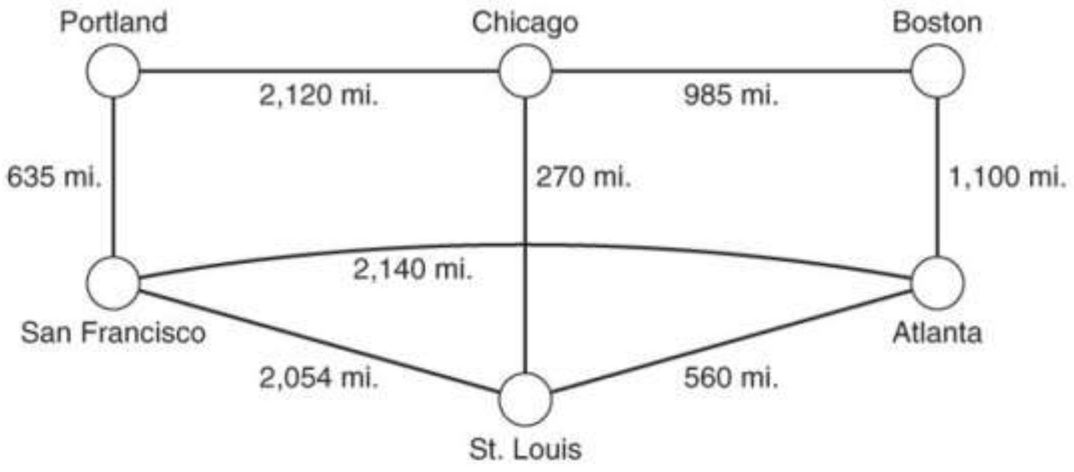


Vertices

- A vertex represents an *entity marked with a unique identifier*—analogous to a row key in a column family database or a primary key in a relational database.
- A vertex can represent virtually any entity that has a relation with another entity.
- Vertices can represent:
  - People in a social network.
  - Cities connected by highways.
  - Proteins that interact with other proteins in the body.
  - Warehouses in a company's distribution network.
- Vertices can have properties.

Edges

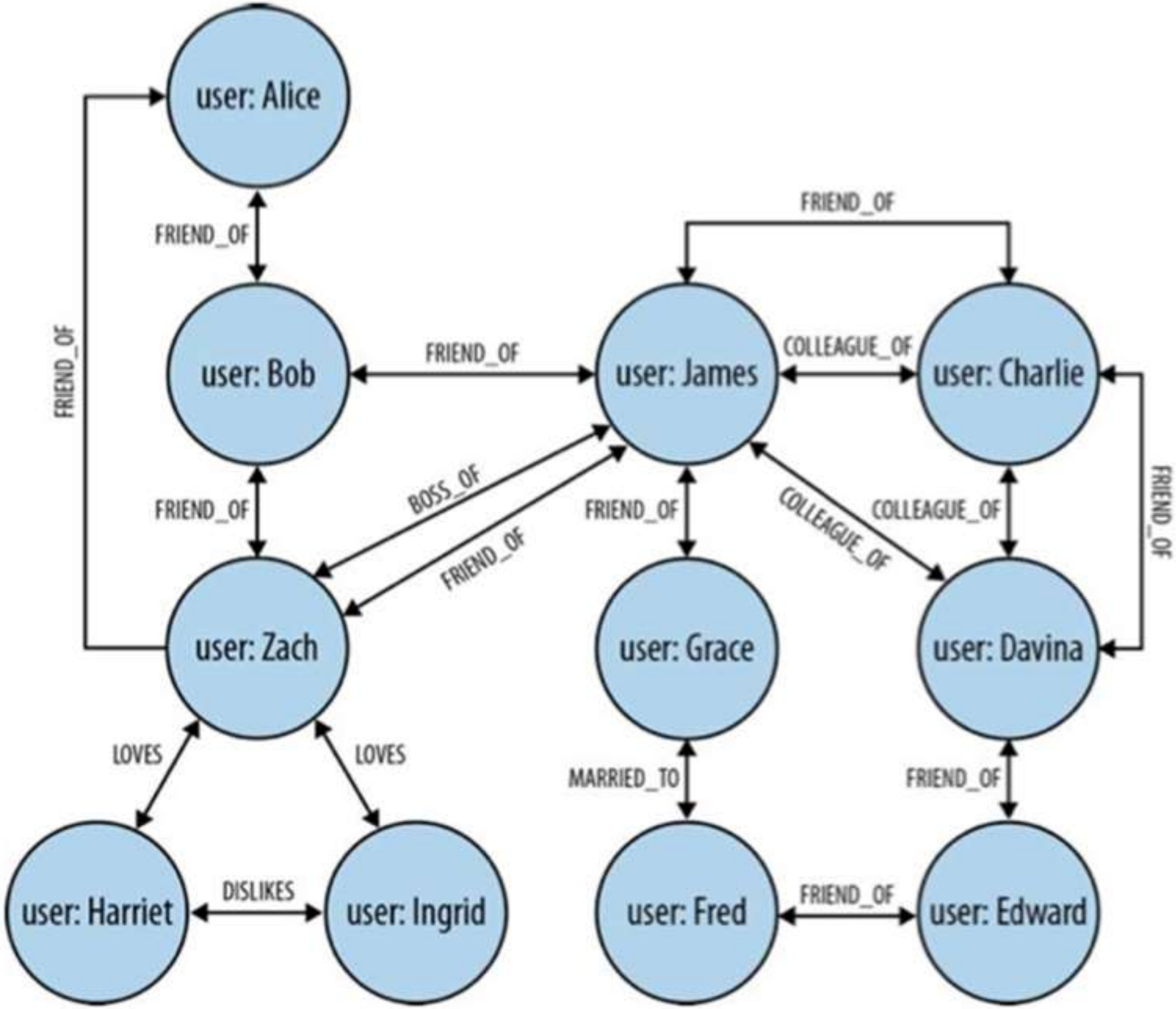
- An edge, also known as *a link* or arc, defines relationships between vertices.
- Much like vertices, edges have *properties*.
- A commonly used property is called the *weight* of an edge. Weights represent some value about the relationship.
- Weights can represent cost, distance, or another measure of the relationship.
- There are two *types of edges*: directed and undirected:
- Directed edges: have a direction. This is used to indicate how the relationship should be interpreted.
- However, direction is not always needed.



Paths

- A path through a graph is a *set of vertices along with the edges* between those vertices. The vertices in a graph are all different from each other.
- If edges are directed, the path is a *directed path*. If the graph is undirected, the paths in it are *undirected paths*.
- Paths are important because they capture information about how vertices in a graph are related.
- A common problem encountered when working with graphs is to find the least weighted path between two vertices.

- Incorporating dynamic information is natural and simple



Native Graph Storage and Processing

- Some GDBMSs use *native graph storage*, which is optimized and designed for storing and managing graphs.
- In contrast to relational DBMSs, these GDBMSs do not store data in disparate tables. Instead, they manage a single data structure.
- Coherently, they adopts a *native graph processing*: they leverage index-free adjacency, meaning that connected nodes physically "point" to each other in the database.

Graph Databases
By Ian Robinson, Jim Webber and Emil Eifrem
O'Reilly Media,

Index-free Adjacency

- A database engine that utilizes index-free adjacency is one in which *each node maintains direct references to its adjacent nodes*; each node, therefore acts as a micro-index of other nearby nodes, which is much cheaper than using global indexes.
- In other terms, a graph database *G* satisfies the index-free adjacency if the existence of an *edge between two nodes v_1 and v_2 in G* can be tested on those nodes and does not require to access an external, global, index.
- *Locally*, each node can manage a *specific index* to speed up access to its outgoing edges

Non-native Graph Storage

- Not all graph database technologies use native graph storage, however. Some serialize the graph data into a *relational database*, *object-oriented databases*, or other types of *general- purpose data stores*.
- GDBMSs of this kind *do not adopt index-free-adjacency*, but uses (global) indexes to link nodes together.
- These indexes add a layer of indirection to each traversal, thereby incurring *greater computational cost.*

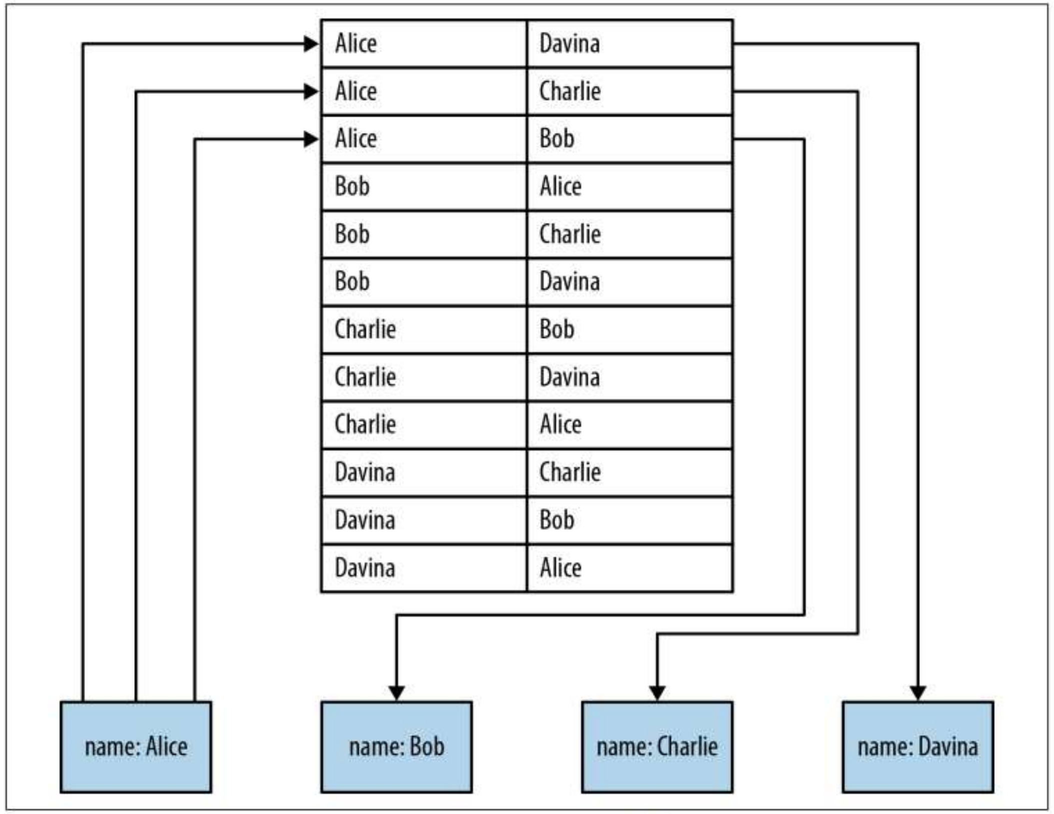
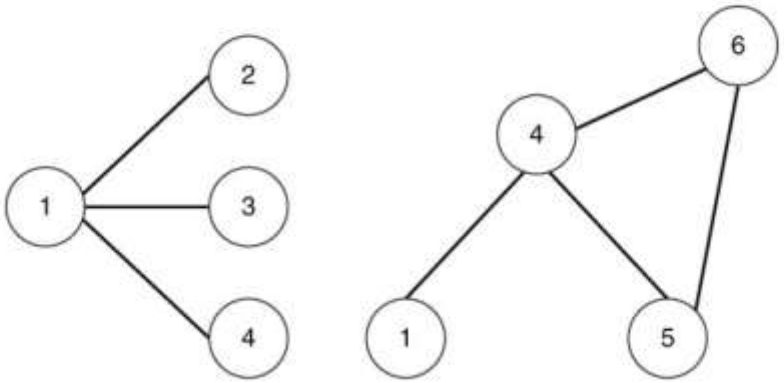


Figure 6-1. Nonnative graph processing engines use indexing to traverse between nodes

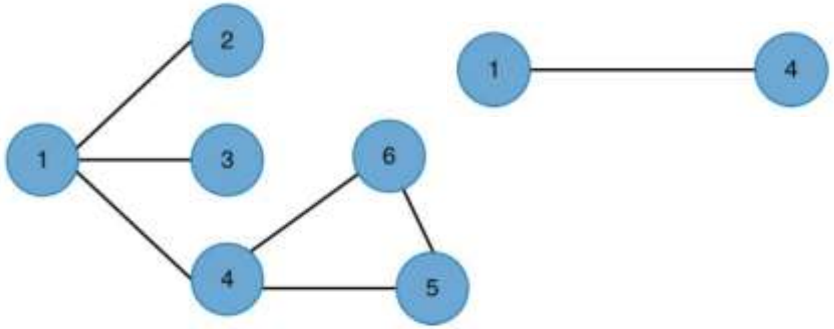
Operations on GraphDB

- All *CRUD operations* like any other database.
- Additional set of operations, specifically, operations can be used to *follow paths* or *detect repeating patterns* in relationships between vertices.
  - Union of graphs
  - Intersection of graphs
  - Graph traversal

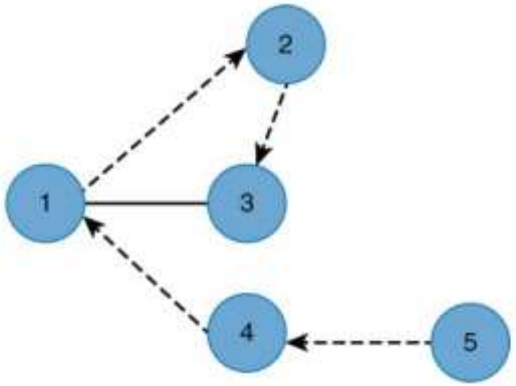
Union, Intersection, and Traverse of Graphs



- The *union* of graphs is the combined set of vertices and edges in a graph.
- The union of A and B is the set of vertices and edges from both graphs.
- Because the two graphs share common vertices, the union produces a single graph



- The *intersection* of a graph is the set of vertices and edges that are common to both graphs.
- Graph *traversal* is the process of visiting all vertices in a graph in a particular way.
- The purpose of this is usually to either set or read some property value in a graph.



Basic Operations

Given a graph G, the following are operations over G:

- AddNode(G,x): adds node x to the graph G.
- DeleteNode(G,x): deletes the node x from graph G.
- Adjacent(G,x,y): tests if there is an edge from x to y.
- Neighbors(G,x): nodes y s.t. there is an edge from x to y.
- AdjacentEdges(G,x,y): set of labels of edges from x to y.

Basic Operations

- Add(G,x,y,l): adds an edge between x and y with label l.
- Delete(G,x,y,l): deletes an edge between x and y with label l.
- Reach(G,x,y): tests if there a path from x to y.
- Path(G,x,y): a (shortest) path from x to y.
- 2-hop(G,x): set of nodes y s.t. there is a path of length 2 from x to y, or from y to x.
- n-hop(G,x): set of nodes y s.t. there is a path of length n from x to y, or from y to x.

Graph DBs vs Relational DBs- Queries*

- Relational Databases (querying is through joins)
  - Joined records are *not explicit* in the relational structure, but instead *must be inferred* through a series of *index-intensive operations*.
  - Moreover, while only a particular subset of the data in the database may be desired (e.g., only Alice's friend's), all data in all queried tables must be examined in order to extract the desired subset

* From: Marko A. Rodriguez, Peter Neubauer: The Graph Traversal Pattern. Graph Data Management 2011: 29-46

Graph DBs vs Relational DBs- Queries*

- Graph Databases (querying is through traversal paths)
  - There is *no explicit join operation* because vertices maintain direct references to their adjacent edges.
  - In many ways, the edges of the graph serve as explicit, “hard-wired” join structures (i.e., structures that are not computed at query time as in a relational database).
  - What makes this more efficient in a graph database is that traversing from one vertex to another is a constant time operation.

* From: Marko A. Rodriguez, Peter Neubauer: The Graph Traversal Pattern. Graph Data Management 2011: 29-46

What are graphs good for?

- Recommendations
- Business intelligence
- Social computing
- Geospatial
- Systems management
- Web of things
- Genealogy

- Time series data
- Product catalogue
- Web analytics
- Scientific computing (especially bioinformatics)
- Indexing your slow RDBMS
- And much more!