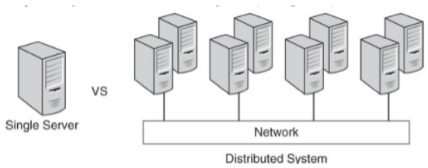
SOMMARIO

Distributed System



Pro:

- Ensure ***scalability***, ***flexibility, cost control*** and ***availability***
- It is easier to add or to remove nodes (***horizontal scalability***) rather than to add memory or to upgrade the CPUs of a single server (***vertical scalability***)
- Allow the implementation of ***fault tolerance*** strategies

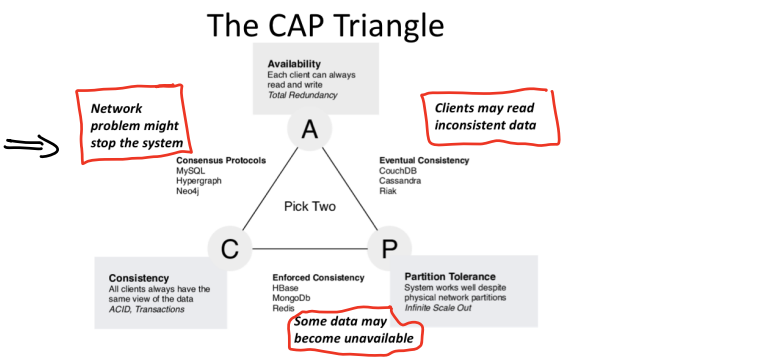
Cons: difficulty

- To ***balance*** the requirements of data ***consistency*** and system ***availability***
- To protect themselves from ***network failures*** that may leave some nodes ***isolated***

CAP Theorem

Distributed Databases cannot ensure at the same time:

- ***Consistency (C)***, the presence of consistent copies of data on different servers
- ***Availability (A)***, namely to providing a response to any query
- ***Partition protection (P)***, Failures of individual nodes or connections between nodes do not impact the system as a whole.



CA -> Choose C and A with compromising of P

Use cases: Banking and Finance application, system which must have transaction e.g. connected to RDBMS.

Site cluster, therefore all nodes are always in contact, when a partition occurs, the system blocks.

AP -> Choose A and P with compromising of C.

Use cases: shopping carts, News publishing CMS, etc.

System is still available under ***partitioning***, but some of the data returned may be inaccurate.

CP -> As suitable for application which require consistency, but also partition tolerance, while somewhat long response times are acceptable (***Bank ATMs***).

NoSQL DBS Base Properties

- ***BA*** stands for ***basically available***: ***partial failures*** of the distributed database may be handled in order to ensure the ***availability*** of the service (often thanks to data ***replication***).
- ***S*** stands for ***soft state***: data stored in the nodes may be ***updated*** with ***more recent*** data because of the eventual consistency model (no user writes may be responsible of the updating!!).
- ***E*** stands for ***eventually consistent***: ***at some point in the future***, data in all nodes will converge to a consistent state.

KEY-VALUE DATABASE

Key-value databases are the simplest of the NoSQL databases.The design of this type of data store is based on storing data with identifiers known as keys.

A key-value data store is a more complex variation on the array data structure. Computer scientists have extended the concept of an array by relaxing constraints on the simple data structure and adding persistent data storage features to create a range of other useful data structures, including associative arrays, caches, and persistent key-value databases.

FROM ARRAY…

1	True
2	True
3	False
4	True
5	False
6	False
7	False
8	True
9	False
10	True

An array is an ordered list of values.

Each value in the array is associated with an integer index.

The values are all the same type.

The index can only be an integer.

The values must all have the same type

… TO ASSOCIATIVE ARRAY …

‘Pi’	3.14
‘CapitalFrance’	‘Paris’
17234	34468
‘Foo’	‘Bar’
‘Start_Value’	1

An associative array is a data structure, like an array, but is not restricted to using integers as indexes or limiting values to the same type.

It generalizes the idea of an ordered list indexed by an identifier to include arbitrary values for identifiers and values.

… TO KEY-VALUE DATABASES

Database

Bucket 1	
‘Foo1’	‘Bar’
‘Foo2’	‘Bar2’
‘Foo3’	‘Bar7’

Bucket 2	
‘Foo1’	‘Baz’
‘Foo4’	‘Baz3’
‘Foo6’	‘Baz2’

Bucket 3	
‘Foo1’	‘Bar7’
‘Foo4’	‘Baz3’
‘Foo7’	‘Baz9’

Key-value data stores are even more useful when they store data persistently on disk, flash devices, or other long-term storage. They offer the **fast performance** benefits of caches plus the **persistent storage** of databases.

Key-value databases impose a minimal set of constraints on how you arrange your data. There is no need for tables if you do not want to think in terms of groups of related attributes.

The one design requirement of a key-value database is that **each value has a unique identifier in the form of the key**. Keys must be unique within the namespace (defined by the key-value database) and it is called **bucket** (collection of key-value pairs).

```
customer:1982737:firstName
customer:1982737:lastName
customer:1982737:shippingAddress
customer:1982737:shippingCity
customer:1982737:shippingState
customer:1982737:shippingZip
```

Could use a key-naming convention that uses a table name, primary key value, and an attribute name to create a key to store the value of an attribute.

Developers tend to use key-value databases when **ease of storage and retrieval are more important than organizing data into more complex data structures**, such as tables or networks.

ESSENTIAL FEATURES OF KEY-VALUE DATABASES

A variety of key-value databases is available to developers, and they all share three essential features:

SIMPLICITY: WHO NEEDS COMPLICATED DATA MODELS ANYWAY?

If we do not need all the features of the relational models (joining tables or running queries about multiple entities in the database), we may exploit key-value databases, we do not need all those extra features.

Microsoft Word, for example, has an impressive list of features, including a wide array of formatting options, spelling and grammar checkers, and even the ability to integrate with other tools like reference and bibliography managers.

Often, developers do not need support for joining tables or running queries about multiple entities in the database. If you were implementing a database to store information about a customer's online shopping cart, you could use a relational database, but it would be simpler to use a key-value database. You would **not have to define a database schema** in SQL. You would **not have to define data types** for each attribute you'd like to track.

If you discover that you would like to track additional attributes after you have written your program, you can simply add code to your program to take care of those attributes. There is no need to change database code to tell the database about the new attribute. Key-value databases have no problem working with **adding new attributes** as they come along.

In key-value databases, you work with a simple data model. The syntax for manipulating data is simple. Typically, you **specify a namespace** (database name), a bucket name, or some other type of collection name, and a key to indicate you want to perform an operation on a key-value pair. To search you specify only the namespace name and the key, the key-value database will return the associated value. When you want to update the value associated with a key, you specify the namespace, key, and new value.

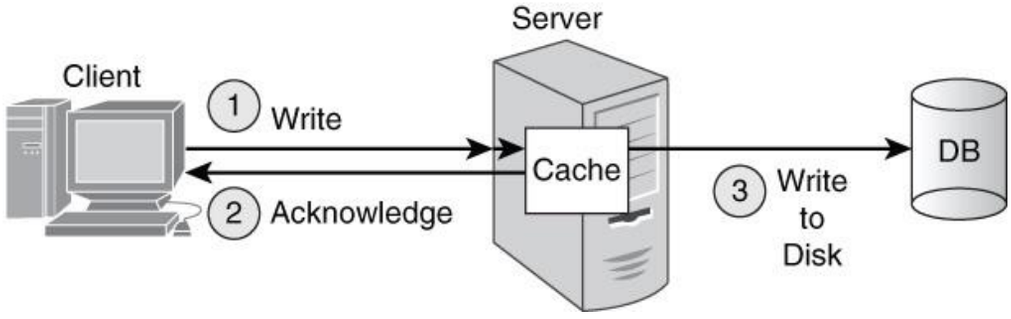
Key-value databases are **flexible** and **forgiving**. If you make a mistake and assign the wrong type of data, for example, a real number instead of an integer, the database usually does not complain. This feature is especially useful when the data type changes or you need to support two or more data types for the same attribute. If you need to have both numbers as strings for customer identifiers, you can do that with code.

SPEED: THERE IS NO SUCH THING AS TOO FAST

Key-value databases are known for their speed. With a simple associative array data structure and design features to optimize performance, key-value databases can deliver high-throughput, data-intensive operations.

One way to keep database operations running fast is to **keep data in memory**. Reading and writing data to RAM is much faster than writing to a disk. Of course, RAM is not persistent storage, so if you lose power on your database server, you will lose the contents of RAM. Key-value databases can have the advantages of fast write operations to RAM and the persistence of disk-based storage by using both.

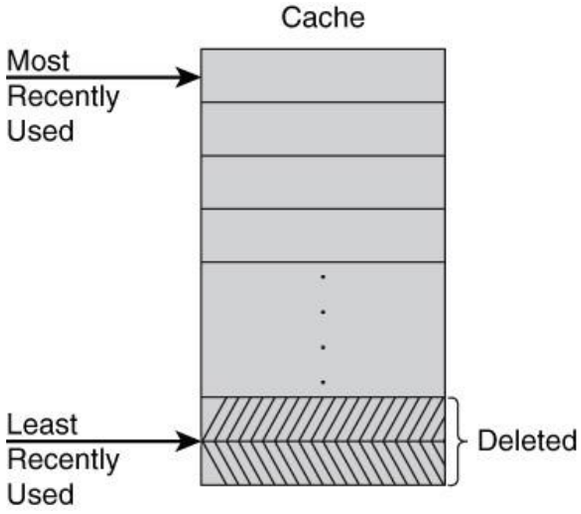
When a program changes the value associated with a key, the key-value database can update the entry in RAM and then send a message to the program that the updated value has been saved. The program can then continue with other operations. While the program is doing something else, the key-value database can write the recently updated value to disk. The new value is saved to disk unless there is a power loss or some other failure between the time the application updates the value and the key-value database stores the value on disk.



Similarly, read operations can be faster if data is stored in memory. This is the motivation for using a cache, as described earlier. Because the size of the database can exceed the size of RAM, key-value stores have to find ways of managing the data in memory.

Compressing data is one way of increasing the effective storage capacity of memory, but even with compression there may not be sufficient memory to store a large key-value database completely in RAM.

THE RAM IS NOT INFINITE!



When the key-value database uses all the memory allocated to it, the database will need to free some of the allocated memory before storing copies of additional data. There are multiple algorithms for this, but a commonly used method is known as least recently used (**LRU**). The idea behind the LRU algorithm is that if data has not been used in a while, it is less likely to be used than data that has been read or written more recently. This intuition makes sense for many application areas of key-value databases.

Use case: Consider a key-value database used to store items in customers' online carts. Assume that once a customer adds an item to the cart, it stays there until the customer checks out or the item is removed by a background cleanup process. A customer who finished shopping several hours ago may still have data in

memory. More than likely, that customer has abandoned the cart and is not likely to continue shopping. Compare that scenario with a customer who last added an item to the cart five minutes ago. There is a good chance that customer is still shopping and will likely add other items to the cart or continue to the checkout process shortly.

SCALABILITY: KEEPING UP WITH THE RUSH

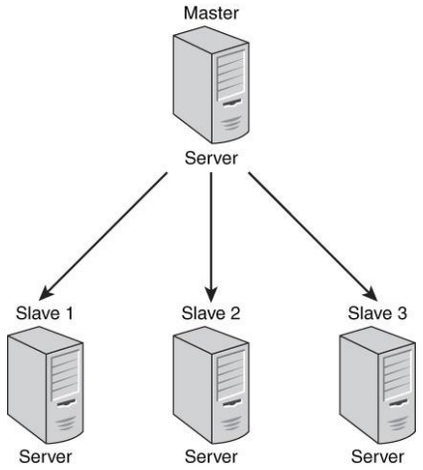
It is important for key-value databases, and other types of NoSQL databases used in web and other large-scale applications, to scale with minimal disruption to operations.

Scalability is the capability **to add or remove servers from a cluster** of servers as needed to accommodate the load on the system. When you scale databases, the **capability to accommodate both reads and writes** is an important property. Key-value databases take different approaches to scaling read and write operations. Let's consider two options:

- Master-slave replication
- Masterless replication

1) SCALING WITH **MASTER-SLAVE** REPLICATION → gestire più letture che scritture

One way to keep up with a growing demand for read operations is to add servers that can respond to queries.



It is easy to imagine applications that would have many more reads than writes. During the World Cup finals, football fans around the world (and soccer fans in the United States) who have to work instead of watch the game would be checking their favorite sport score website for the latest updates. News sites would similarly have a greater proportion of reads than writes. Even e-commerce sites can experience a higher ratio of page views than data writes because customers may browse many descriptions and reviews for each item they ultimately end up adding to their shopping carts.

In applications such as this, it is reasonable to have more servers that can respond to queries than accept writes. A master-slave replication model works well in this case. The master is a server in the cluster that accepts write and read requests. It is responsible for maintaining the master record of all writes and replicating, or copying, updated data to all other servers in the cluster. These other servers only respond to read requests (master-slave architectures have a simple hierarchical structure).

- **Advantage**

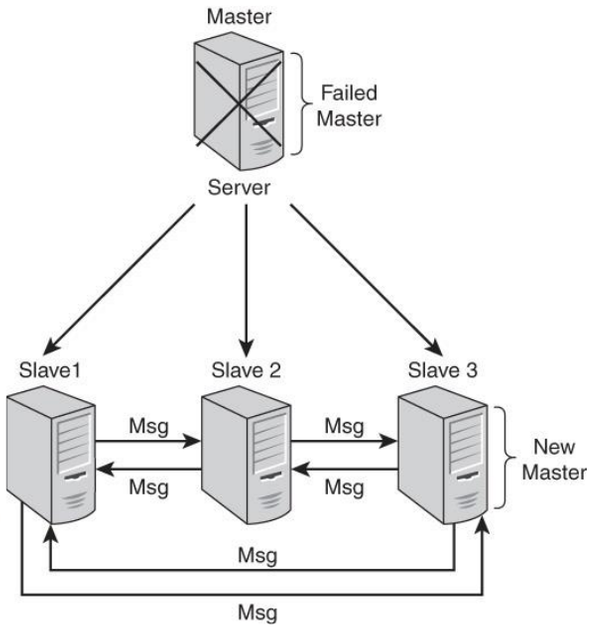
Simplicity. Except for the master, each node in the cluster only needs to communicate with one other server: the master. The master accepts all writes, so there is no need to coordinate write operations or resolve conflicts between multiple servers accepting writes.

- **Disadvantage**

Single point of failure. If the master fails, the cluster cannot accept writes. This can adversely impact the availability of the cluster, that is, a single component in a system that if it fails, the entire system fails or at least loses a critical capacity, such as accepting writes.

Designers of distributed systems have developed protocols so active servers can detect when other servers in the cluster fail. For example, a server may send a simple message to ask a random server in the cluster if it is still active. If the randomly selected server replies, then the first server will know the other server is active.

In the case of master-slave configurations, if a number of slave servers do not receive a message from the master within some period of time, the slaves may determine the master has failed. At that point, the slaves initiate a protocol to promote one of the slaves to master. Once active as the master, the new master server begins accepting write operations and the cluster would continue to function, accepting both read and write operations.



2) SCALING WITH **MASTERLESS** REPLICATION => Gestire sia molte letture che scritture

The master-slave replication model with a single server accepting writes does not work well when there are a large number of writes. Imagine the Rolling Stones decide to have one more world tour. Fans around the world flock to buy concert tickets. The fans would generate a large number of reads when they look up the cities that will be hosting concerts, but once they find one or two close cities, they are ready to purchase tickets.

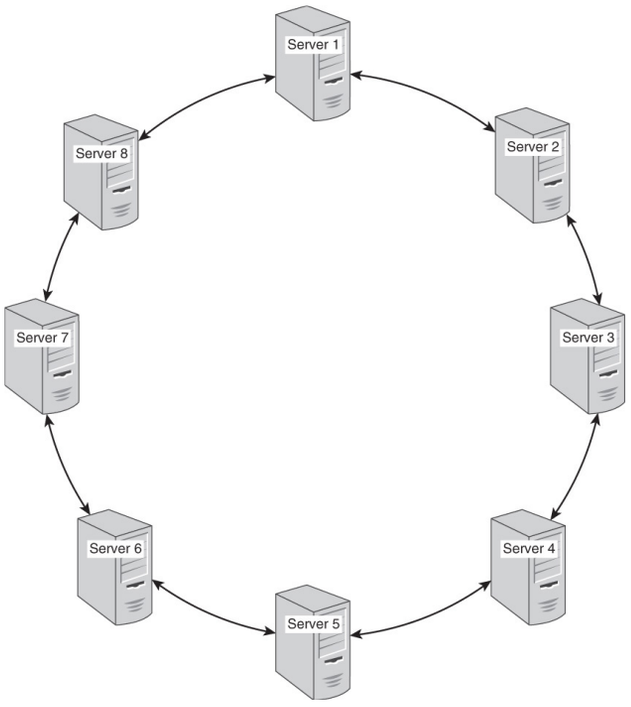
The software engineers who write the concert ticket program have a lot to think about, including: storing concert locations and dates, available seats in each venue, cost of seats in various sections, any limits on the number of tickets purchased by a single customer, ensuring that seats that appear to be available to a user are still available when the user chooses to purchase the ticket. This assumes the customer opts to buy the ticket almost immediately after seeing the availability.

With the possibility of a surge in the number of customers trying to write to the database, a single server accepting writes will limit scalability. A better option for this application is a masterless replication model in which all nodes accept reads and writes. An immediate problem that comes to mind is: How do you handle writes so that two or more servers do not try to sell the same seat in a concert venue to multiple customers?

We will see an elegant solution in the next paragraph.

For now, let's assume that only one customer can purchase a seat at a concert venue at a particular date and time. There is still the problem of scaling reads.

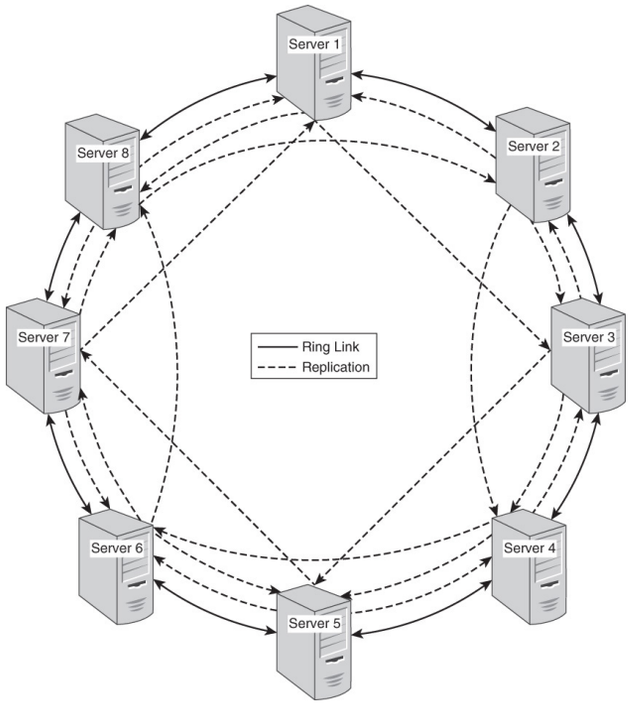
In a masterless replication model, there is not a single server that has the master copy of updated data, so no single server can copy its data to all other servers. Instead, servers in a masterless replication model work in groups to help their neighbors.



Consider a set of eight servers configured in a masterless replication model and set up in a ring structure. In the ring structure, Server 1 is logically linked to Servers 2 and 8, Server 2 is linked to Servers 1 and 3, Server 3 is linked to Servers 2 and 4, and so on.

It's used a

ring structure is a useful abstraction for discussing replication in a masterless model. In a data center, the eight servers would probably all be connected to a single network hub and able to directly communicate with each other.



Database administrators can configure a key-value database to keep a particular number of replicas. In this scenario, the administrator has decided that four replicas are sufficient.

Each time there is a write operation to one of the servers, it replicates that change to the three other servers holding its replica. In this scenario, each server replicates to its two neighbors and to the server two links ahead. For example, Server 2 replicates to its neighbors Server 1 and Server 3 as well as Server 4, which is two links ahead.

For each write there is a propagation of the data.

HOW TO CONSTRUCT A KEY

Keys are used to identify, index, or otherwise reference a value in a keyvalue database. The one essential property of a key is that it must be **unique within a namespace**.

In relational model, the primary key is used to retrieve a row in a table and it must be unique. Designers could use one counter to generate primary keys for all tables, or they could use a different counter or sequence for each table. Either way, each row in a table has a unique identifier. When designing relational databases, using meaningless primary keys is a good practice.

Just as keys in key-value databases must be unique in a namespace, the primary key of a row of data must be unique to the table and key is the unique mean to retrieve a value. In the framework of key-value databases, adopting meaningful keys is preferable.

The sole purpose of a primary key, is to uniquely identify a row. If you were to use a property of the data, such as the last name and first initial of a customer, you might run into problems with duplicates. Also, values stored in rows may change.

For example, consider how quickly the meaning of a primary key would change if you used the two-letter state abbreviation of the state in which a customer lives as part of a key for that customer. You could have a key such as 'SMITH_K_TX' for a Katherine Smith who lives in Texas. If Katherine Smith moves to Illinois, then the primary key is no longer meaningful. Same for the email.

Primary keys should not be changed, so you could not simply change the key to 'SMITH_K_IL.' That would violate the principle that **primary keys are immutable**. You could conceivably change a primary key (if the database management system allowed such updates), but you would have to update all references to that key in other tables. Thus, it is important to select PK without specific meanings.

In NoSQL databases, and key-value databases in general, the rules are different. Key-value databases do **not have a built-in table structure**. With no tables, there are no columns. With no columns, there is no way to know what a value is for except for the key.

Consider a shopping cart application using a key-value database with meaningless keys:

Cart[12387] = 'SKU AK8912j4'

This key is the type of identifier you would likely see in a relational database. This key-value pair only tells you that a cart identified by number 12387 has an item called 'SKU AK8912j4'. You might assume from the value that SKU stands for stock keeping unit, a standard term in retail to refer to a specific type of product. However, you don't know who this cart belongs to or where to ship the product.

One way to solve this problem is to create another namespace, such as custName. Then you could save a value such as

CustName[12387] = 'Katherine Smith'

This would solve the immediate problem of identifying who owns the cart, but you can see that this approach does not generalize well. Every attribute tracked in the application would need a separate namespace. Alternatively, you can use meaningful keys that entail information about attributes.

As discussed earlier, you can construct meaningful names that entail information aboutentity types, entity identifiers, and entity attributes. For example:

Cust:12387:firstName

could be a key to store the first name of the customer with customerID 12387 (This is not the only way to create meaningful names). Again, the basic formula is

Entity Name + ':' + Entity Id +':' + Entity Attribute es: Customer : 1234: FirstName
Customer : 1234: LastName

USING KEYS TO LOCATE VALUES

Up to this point, there has been a fair amount of discussion about how to construct keys, why keys must be unique within a namespace, and why **meaningful** keys are more useful in key-value databases than relational

databases. There has been some mention of the idea that keys are used to look up associated values, but there has been no explanation about how that happens. It is time to address that topic.

If key-value database designers were willing to restrict you to using integers as key values, then they would have an easy job of designing code to fetch or set values based on keys.

They could load a database into memory or store it on disk and assume that the first value stored in a namespace is referenced by key 1, the next value by key 2, and so on.

Fortunately, key-value designers are more concerned with designing useful data stores than simplifying data access code. Using numbers to identify locations is a good idea, but it is not flexible enough. You should be able to use integers, character strings, and even lists of objects as keys if you want. The good news is that you can. The trick is to use a function that maps from integers, character strings, or lists of objects to a unique string or number. These functions that map from one type of value to a number are known as **hash functions**.

- **Hash Functions: From Keys to Locations**

A hash function is a function that can take an arbitrary string of characters and produce a (usually) unique, fixed-length string of characters. Actually, the value returned by the hash function is not always unique; sometimes two unrelated inputs can generate the same output. This is known as a collision.

Key	Hash Value
customer:1982737: firstName	e135e850b892348a4e516cfcb385eba3bfb6d209
customer:1982737: lastName	f584667c5938571996379f256b8c82d2f5e0f62f
customer:1982737: shippingAddress	d891f26dcdb3136ea76092b1a70bc324c424ae1e
customer:1982737: shippingCity	33522192da50ea66bfc05b74d1315778b6369ec5
customer:1982737: shippingState	239ba0b4c437368ef2b16ecf58c62b5e6409722f
customer:1982737: shippingZip	814f3b2281e49941e1e7a03b223da28a8e0762ff

How we can see the keys, in key value database, are meaningful because they show what they what to represent.

In the example above, the Hash Value is a number in hexadecimal format

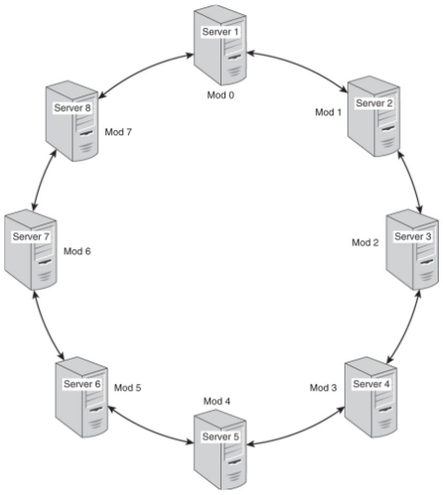
Each hash value is quite different from the others, although they all have the same 'customer:1982737:' prefix. One of the properties of hash functions is that they map to what appear to be random outputs. In this example, the SHA-1 hash function is used to generate the hash values.

KEYS HELP AVOID WRITE PROBLEMS

Now, let's see how you can use the numbers returned by the hash function to map to a location. To keep things simple, the discussion focuses on using the number returned by a hash function to **determine which server in a cluster should be used to store the value associated with the key**. We would like to balance the write loads among the servers in a masterless replication architecture.

Assume you are working with an eight-server cluster. You can take advantage of the fact that the hash function returns a number. Because the write load should be evenly distributed across all eight servers, you can send one eighth of all writes to each server. You could send the first write to Server 1, the second to

Server 2, the third to Server 3, and so on in a round-robin fashion, but this would not take advantage of the hash value.



One way to take advantage of the hash value is to start by dividing the hash value by the number of servers. Sometimes the hash value will divide evenly by the number of servers. (For this discussion, assume the hash function returns decimal numbers, not hexadecimal numbers, and that the number of digits in the number is not fixed.)

If the hash function returns the number 32 and that number is divided by 8, then the remainder is 0. If the hash function returns 41 and it is divided by 8, then the remainder is 1. If the hash function returns 67, division by 8 leaves a remainder of 3. As you can see, any division by 8 will have a remainder between 0 and 7. Each of the eight servers can be assigned a number between 0 and 7 (**MODULE**).

Let's return to the concert ticket application. A challenge was to ensure that two servers did not sell tickets to the same seat, at the same venue, in the same city, on the same night to more than one person. Because key-value databases running in a masterless configuration can accept writes from all servers, such a mistake could happen. The solution is to make sure any requests for the same seat, at the same venue, in the same city, on the same night all go to the same server.

You can do this by making a key based on seat, venue, city, and date. For example, two fans want to purchase set A73 at the Civic Center in Portland, Oregon, on July 15. You could construct keys using the seat, an abbreviation for the venue (CIvCen in this case), the airport code for the city (PDX in this case), and a four-digit number for the date. In this example, the key would be

A73:CivCen:PDX:0715

Anyone trying to purchase that same seat on the same day would generate the same key. Because keys are mapped to servers using modulo operations, all requests for that seat, location, and date combination would go to the same server. There is no chance for another server to sell that seat, thus avoiding the problem with servers competing to sell the same ticket.

Keys, of course, are only half the story in key-value databases. It is time to discuss values.

VALUES DO NOT REQUIRE STRONG TYPING

Unlike strongly typed programming languages that require you to define variables and specify a type for those variables, key-value databases do not expect you to specify types for the values you store.

You could, for example, store a string along with a key for a customer's address:

String:'1232 NE River Ave, St. Louis, MO'

or you could store a list of the form:

List of strings: ('1232 NE River Ave', 'St. Louis','MO')

or you could store a more structured format using JavaScript Object Notation, such as

JSON format: {'Street:' : '1232 NE River Ave',

'City' : 'St. Louis',:

'State' : 'MO' }

Key-value databases make minimal assumptions about the structure of data stored in the database.

While in theory, key-value databases allow for arbitrary types of values, in practice database designers have to make implementation choices that lead to some restrictions. Different implementations of key-value databases have different restrictions on values.

For example, some key-value databases will typically have some limit on the size of values. Some might allow multiple megabytes in each value, but others might have smaller size limitations.

Even in cases in which you can store extremely large values, you might run into performance problems that lead you to work with smaller data values.

It is important to consider the design characteristics of the key-value database you choose to use. Consult the documentation for limitations on keys and values. Part of the process in choosing a key-value database is considering the trade-off of various features. One key-value database might offer ACID transactions but limit you to small keys and values. Another key-value data store might allow for large values but limit keys to numbers or strings. Your **application requirements** should be considered when weighing the **advantages** and **disadvantages** of different database systems.

LIMITATIONS ON SEARCHING FOR VALUES

Keep in mind that in key-value databases, operations on values are all based on keys. You can **retrieve** a value by key, you can **set** a value by key, and you can **delete** values by key.

That is pretty much the repertoire of operations. If you want to do more, such as search for an address in which the city is "St. Louis," you will have to do that with an application program.

Key-value databases do not support query languages for searching over values. There are two ways to address this limitation.

You, as an application developer, could implement the required search operations in your application. For example, you could generate a series of keys, query for the value of each key, and test the returned value for the pattern you seek.

Let's assume you decided to store addresses as a string such as '1232 NE River Ave, St. Louis, MO' and you store it like this:

appData[cust:9877:address] = '1232 NE River Ave, St. Louis, MO'

```
define findCustomerWithCity(p_startID, p_endID, p_City):
    begin
    # first, create an empty list variable to hold all
    # addresses that match the city name
    returnList = ();
    # loop through a range of identifiers and build keys
    # to look up address values then test each address
    # using the inString function to see if the city name
    # passed in the p_City parameter is in the address
    # string. If it is, add it to the list of  addresses
    # to return
    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);
end;
```

This method enables you to search value strings, but it is inefficient. If you need to search large ranges of data, you might retrieve and test many values that do not have the city you are looking for.

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'
	. . . .
'Portland'	'cust:9134:city'
'Chicago'	'cust:2149:city', 'cust:4111:city'

Some key-value databases incorporate search functionality directly into the database. This is an additional service not typically found in key-value databases but can significantly add to the usefulness of the database. A built-in search system would index the string values stored in the database and create an index for rapid retrieval. Rather than search all values for a string, the search system keeps a list of words with the keys of each key-value pair in which that word appears.

KEY

A key is a reference to a value. It is analogous to an address. The address 1232 NE River St. is a reference to a building located in a particular place. Among other things, it enables postal workers and delivery services to find a particular building and drop off or pick up letters and packages. The string “1232 NE River St.” is obviously not a building, but it is a way to find the corresponding building. Keys in key-value databases are similarly not values but are ways of finding and manipulating values.

A key can take on different forms depending on the key-value database used. At a minimum, a key is specified as a string of characters, such as "Cust9876" or "Patient:A384J:Allergies". Some key-value databases, such as Redis (www.redis.io), support more complex data structures as keys. The supported key data types in Redis version 2.8.13 include: Strings, Lists, Sets, Sorted sets, Hashes, Bit arrays.

Lists are ordered collections of strings. Sets are collections of unique items in no particular order. Sorted sets, as the name implies, are collections of unique items in a particular order. Hashes are data structures that have key-value characteristics: They map from one string to another. Bit arrays are binary integer arrays in which each individual bit can be manipulated using various bit array operations.

Keep in mind that strings should not be too long. Long keys will use more memory and key-value databases tend to be memory-intensive systems already. At the same time, avoid keys that are too short. Short keys are more likely to lead to conflicts in key names. For example, the key CMP:1897:Name could refer to the name of a marketing campaign or the name of a component in a product. A better option would be CAMPN:1897:Name to refer to a marketing campaign and COMPT:1897:Name to refer to a component in a product.

VALUE

The definition of value with respect to key-value databases is so amorphous that it is almost not useful. A value is an object, typically 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), semistructured constructs such as JSON objects, images, audio, and just about any other data type you can represent as a series of bytes.

It is important to understand that different implementations of key-value databases have different restrictions on values. Most key-value databases will have a limit on the size of a value. Redis, for example, can have a string value up to 512MB in length. FoundationDB (foundationdb.com), a key-value database known for its support of ACID transactions, limits the size of values to 100,000 bytes.

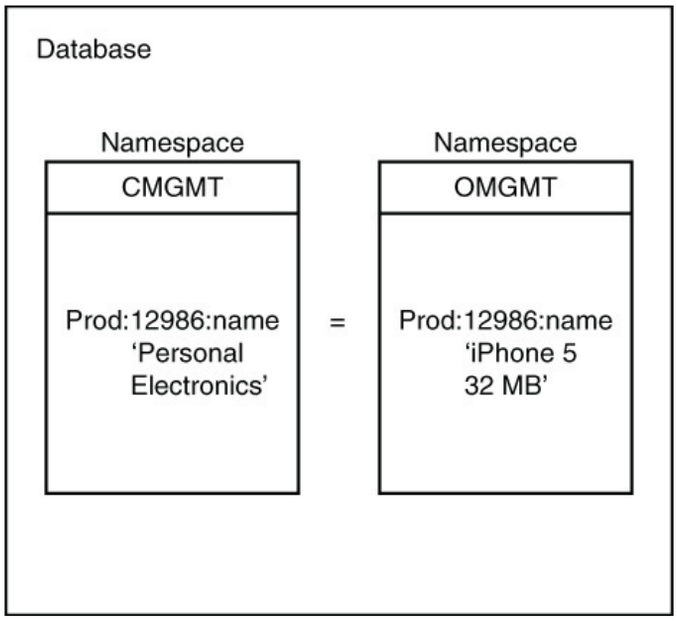
Key-value implementations will vary in the types of operations supported on values. At the very least, a key-value database will support getting and setting values. Others support additional operations, such as appending a string to an existing value or randomly accessing a section of a string. This can be more efficient than retrieving a value, returning it to a client application, performing the append operation in the client application, and then performing a set operation to update the value.

Limits on the dimension of a single value may be fixed by the different frameworks for Key-Value databases.

NAMESPACE

A namespace is a collection of key-value pairs. You can think of a namespace as a set, a collection, a list of key-value pairs without duplicates, or a bucket for holding key-value pairs. A namespace could be an entire key-value database. The essential characteristic of a namespace is it is a collection of key-value pairs that has no duplicate keys. It is permissible to have duplicate values in a namespace.

Namespaces are helpful when multiple applications use a key-value database. Developers of different applications should not have to coordinate their key-naming strategy unless they are sharing data.



For example, one development team might work on a customer management system while another is working on an order-tracking system. Both will need to use customers' names and addresses. In this case, it makes sense to have a single set of customers used by both teams. It would avoid duplicate work to maintain two customer lists and eliminate the possibility of inconsistent data between the two databases.

When the two teams need to model data specific to their application, there is a potential for key-naming conflicts. The team working on the customer management system might want to track the top type of products each customer purchases, for example, personal electronics, clothing, sports, and so on. The team decides to use the prefix Prod for their product type keys. The team working on order tracking also needs to track products but at a more detailed level. Instead of tracking broad categories, like personal electronics, they track specific products, such as an iPhone 5 32MB. They also decide to use the prefix Prod.

You can probably see the problem this raises. Imagine both applications use the same customer data and, therefore, customer IDs. The customer management team might create a key such as 'Prod:12986:name' and assign the value 'personal electronic.' Meanwhile, the order management team wants to track the last product ordered by a customer and creates the key 'Prod:12986:name' and assigns it the value 'iPhone 5 32MB.'

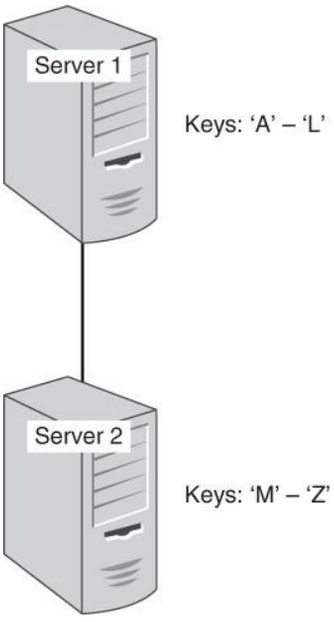
In this situation, the value of the key is set to the last value written by one of the applications. When the other application reads the data, it will find not only an incorrect value, but also one that is out of the range of expected values.

Namespaces solve this problem by implicitly defining an additional prefix for keys. The customer management team could create a namespace called custMgmt, and the order management team could create a namespace called ordMgmt. They would then store all keys and values in their respective

namespaces. The key that caused problems before effectively becomes two unique keys: custMgmt: Prod:12986:name and ordMgmt: Prod:12986:name.

PARTITION

Just as it is helpful to organize data into subunits—that is, namespaces—it is also helpful to organize servers in a cluster into subunits. A partitioned cluster is a group of servers in which servers or instances of key-value database software running on servers are assigned to manage subsets of a database.

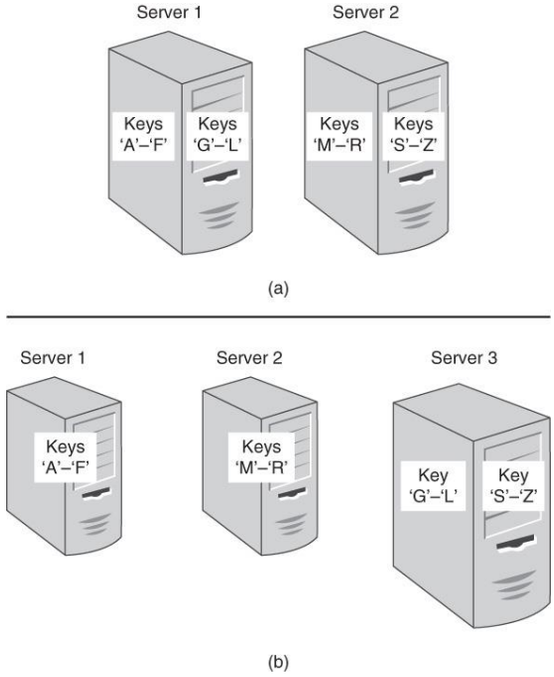


Let's consider a simple example of a two-server cluster. Each server is running key-value database software. Ideally, each server should handle 50% of the workload. There are several ways to handle this. You could simply decide that all keys starting with the letters A through L are handled by Server 1 and all keys starting with M through Z are managed by Server 2. (Assume for the moment that all keys start with a letter.) In this case, you are partitioning data based on the first letter of the key.

Like so many simple strategies that sound reasonable at first, this one is vulnerable to significant problems. For example, most of the keys may start with the letter C, as in cust (customer), cmpg (campaign), comp (component), and so on, whereas very few keys start with letters from the latter half of the alphabet, for example, warh (warehouse). This imbalance in keys leads to an imbalance in the amount of work done by each server in the cluster.

Partition schemes should be chosen to distribute the workload as evenly as possible across the cluster. The "Partition Key" section describes a widely used method to help ensure a fairly even distribution of data and, therefore, workloads.

Note that a server may support more than one partition. This can happen if servers are running virtual machines and each virtual machine supports a single partition. Alternatively, key-value databases may run multiple instances of partition software on each server. This allows for a number of partitions larger than the number of servers.



PARTITION KEY

A partition key is a key used to determine which partition should hold a data value. In keyvalue databases, all keys are used to determine where the associated value should be stored. Later, you see that other NoSQL database types, such as document databases, use one of several attributes in a document as a partition key.

In the previous example, the first letter of a key name is used to determine which partition manages it. Other simple strategies are partitioning by numeric value and string value. Any key in a key-value database is used as a partition key; good partition keys are ones that distribute workloads evenly.

In some cases, you may not have a key that by itself naturally distributes workloads evenly. In these cases, it helps to use a **hash function**. Hash functions map an input string to a fixed-sized string that is usually unique to the input string.

Key, value, namespace, partition, and partition key are all constructs that help you organize data within a key-value database. The key-value database software that you use makes use of particular architectures, or arrangements of hardware and software components. It is now time to describe important terms related to key-value database architecture.

SCHEMALESS

Schemaless is a term that describes the logical model of a database. In the case of key-value databases, you are not required to define all the keys and types of values you will use prior to adding them to the database. If you would like to store a customer name as a full name using a key such as

cust:8983:fullName = 'Jane Anderson'

you can do so without first specifying a description of the key or indicating the data type of the values is a string. Schemaless data models allow you to make changes as needed without changing a schema that catalogs all keys and value types.

Key-Value Database	
Keys	Values
cust : 8983 : firstName	'Jane'
cust : 8983 : lastName	'Anderson'
cust : 8983 : fullName	'Jane Anderson'

We may decide to change how storing the attributes of a specific entity.

For example, you might decide that storing a customer's full name in a single value is a bad idea. You conclude that using separate first and last names would be better. You could simply change your code to save keys and values using statements such as the following:

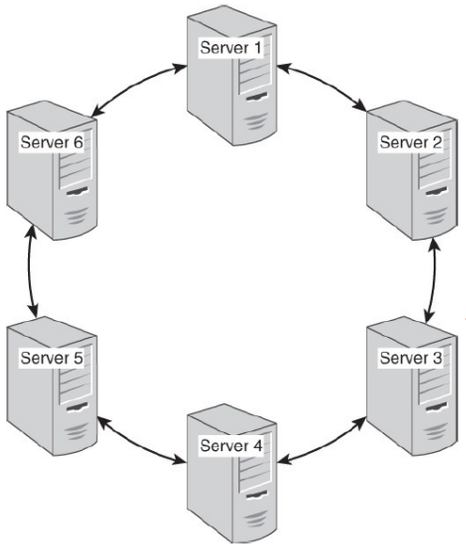
cust:8983:firstName = 'Jane'

cust:8983:lastName = 'Anderson'

The full name and first/last name keys and values can coexist without a problem.

We need to update the application code to handle both ways of representing customer names or convert all instances of one form into the other.

CLUSTER



Clusters are sets to connected computers that coordinate their operations.

Clusters may be loosely or tightly coupled. Loosely coupled clusters consist of fairly independent servers that complete many functions on their own with minimal coordination with other servers in the cluster. Tightly coupled clusters tend to have high levels of communication between servers. This is needed to support more coordinated operations, or calculations, on the cluster. **Key-value clusters tend to be loosely coupled**.

Servers, also known as nodes, in a loosely coupled cluster share information about the range of data the server is responsible for and routinely **send messages to each other to indicate they are still functioning**. The latter message exchange is used to detect failed nodes. When a node fails, the other nodes in the cluster can respond by taking over the work of that node.

Some clusters have a master node. The master node in Redis, for example, is responsible for accepting read and write operations and copying, or replicating, copies of data to slave nodes that respond to read requests. If a master node fails, the remaining nodes in the cluster will elect a new master node. If a slave node fails, the other nodes in the cluster can continue to respond to read requests.

Masterless clusters, such as used by Riak, have nodes that all carry out operations to support read and write operations. If one of those nodes fails, other nodes will take on the read and write responsibilities of the failed node. Because the failed node was also responsible for writes, the nodes that take over for the failed node must have copies of the failed node's data. Ensuring there are multiple copies of data on different nodes is the responsibility of the replication subsystem. This is described in the section "Replication," later in this chapter.

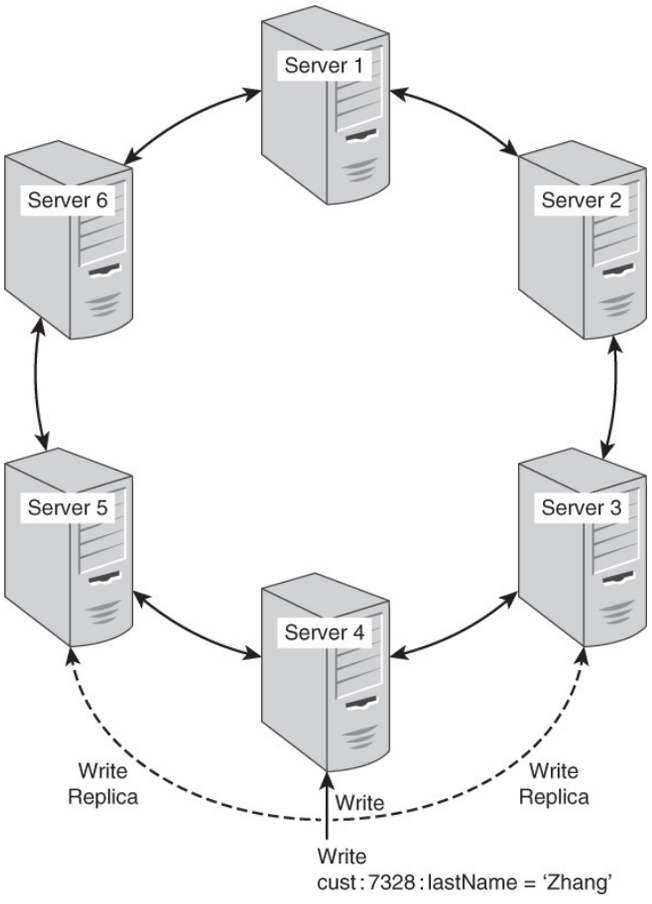
Each node in a masterless cluster is responsible for managing some set of partitions. One way to organize partitions is in a ring structure.

RING

A ring is a logical structure for organizing partitions. A ring is a circular pattern in which each server or instance of key-value database software running on a server is linked to two adjacent servers or instances. Each server or instance is responsible for managing a range of data based on a partition key.

Server Name	Partition Range
Server 1	0–11
Server 2	12–23
Server 3	24–35
Server 4	36–47
Server 5	48–59
Server 6	60–71
Server 7	72–83
Server 8	84–95

Consider a simple hashlike function that maps a partition key from a string; for example, 'cust:8983:firstName' to a number between 0 and 95. Now assume that you have an eight-node cluster and the servers are labeled Server 1, Server 2, Server 3, and so on. With eight servers and 96 possible hashlike values, you could map the partitions to servers..



In this model, Server 2 is linked to Server 1 and Server 3; Server 3 is linked to Server 2 and Server 4; and so on. Server 1 is linked to Server 8 and Server 2.

A ring architecture helps to simplify some otherwise potentially complex operations. For example, whenever a piece of data is written to a server, it is also written to the two servers linked to the original server (**high availability**). This enables high availability of a key-value database. For example, if Server 4 fails, both Server 3 and Server 5 could respond to read requests for the data on Server 4. Servers 3 and 5 could also accept write operations destined for Server 4. When Server 4 is back online, Servers 3 and 5 can update Server 4 with the writes that occurred while it was down.

REPLICATION

Replication is the process of saving multiple copies of data in the nodes of cluster. This provides for high availability as described previously.

One parameter you will want to consider is the number of replicas to maintain. The more replicas you have, the less likely you will lose data; however, you might have lower performance with a large number of replicas. If your data is easily regenerated and reloaded into your key-value database, you might want to use a small number of replicas.

If you have little tolerance for losing data, a higher replica number is recommended. Some NoSQL databases enable you to specify how many replicas must be written before a write operation is considered complete from the perspective of the application sending the write request. For example, you may configure your database to store three replicas. You may also specify that as soon as two of the replicas are successfully written, a successful write return value can be sent to the application making the write request. The third replica will still be written, but it will be done while the application continues to do other work.

You should take replicas into consideration with reads as well. Because key-value databases do not typically enforce two-phase commits, it is possible that replicas have different versions of data. All the versions will eventually be consistent, but sometimes they may be out of sync for short periods.

To minimize the risk of reading old, out-of-date data, you can specify the number of nodes that must respond with the same answer to a read request before a response is returned to the calling application. If you are keeping three replicas of data, you may want to have at least two responses from replicas before issuing a response to the calling program.

The higher the number required, the more likely you are to send the latest response. This can add to the latency of the read because you might have to wait longer for the third server to respond.

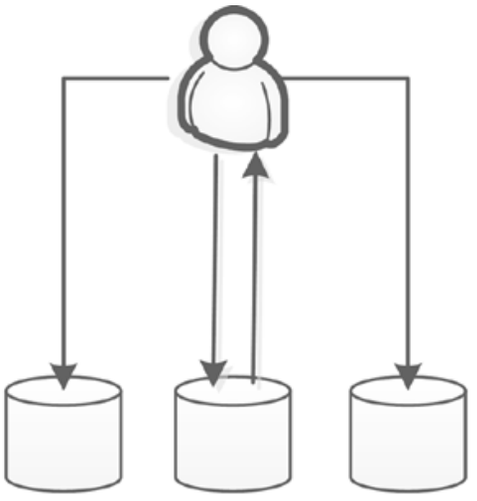
Dynamo (Amazon's key-value database) allows the application to choose the level of consistency applied to specific operations. NWR notation describes how Dynamo will trade off consistency, read performance, and write performance:

- N is the number of copies of each data item that the database will maintain.
- W is the number of copies of the data item that must be written before the write can complete.
- R is the number of copies that the application will access when reading the data item.

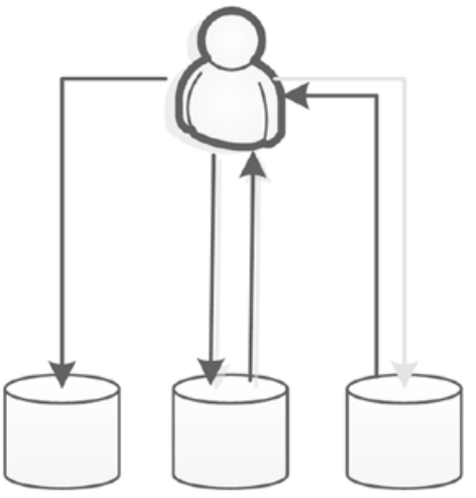
When W = N, Dynamo will always write every copy before returning control to the application—this is what ACID databases do when implementing synchronous replication. If the application is more concerned about write performance than read performance, then it could set W = 1, R = N. Then each read must access all copies to determine which is correct, but each write only has to touch a single copy of the data before returning control (other writes propagate to all copies as a background task).

Probably the most common configuration is N > W > 1. More than one write must complete, but not all nodes need to be updated immediately. Another common setting is W + R > N; this ensures that the latest value will always be included in a read operation, even if it is mixed in with "older" values. This is sometimes referred to as quorum assembly.

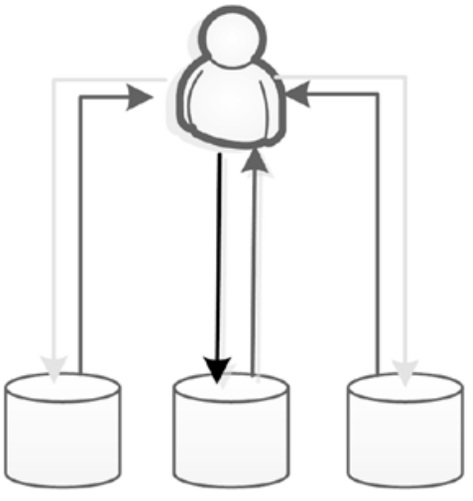
Depending on the settings, Dynamo can trade off consistency, reliability, and performance.



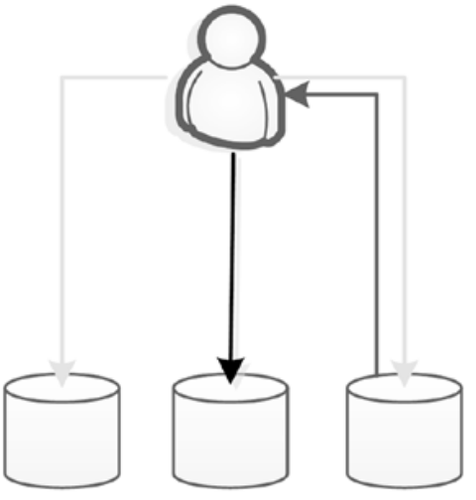
N=3 W=3 R=1
Slow writes, fast reads, consistent
There will be 3 copies of the data.
A write request only returns when all 3 have written to disk.
A read request only needs to read one version.



N=3 W=2 R=2
Faster writes, still consistent (quorum assembly)
There will be 3 copies of the data.
A write request returns when 2 copies are written – the other can happen later.
A read request reads 2 copies make sure it has the latest version.



N=3 W=1 R=N
Fastest write, slow but consistent reads
There will be 3 copies of the data.
A write request returns once the first copy is written – the other 2 can happen later.
A read request reads all copies to make sure it gets the latest version.
Data might be lost if a node fails before the second write.



N=3 W=1 R=1
Fast, but not consistent
There will be 3 copies of the data.
A write request returns once the first copy is written – the other 2 can happen later.
A read request reads a single version only: it might not get the latest copy.
Data might be lost if a node fails before the second write.

CONSISTENT HASHING

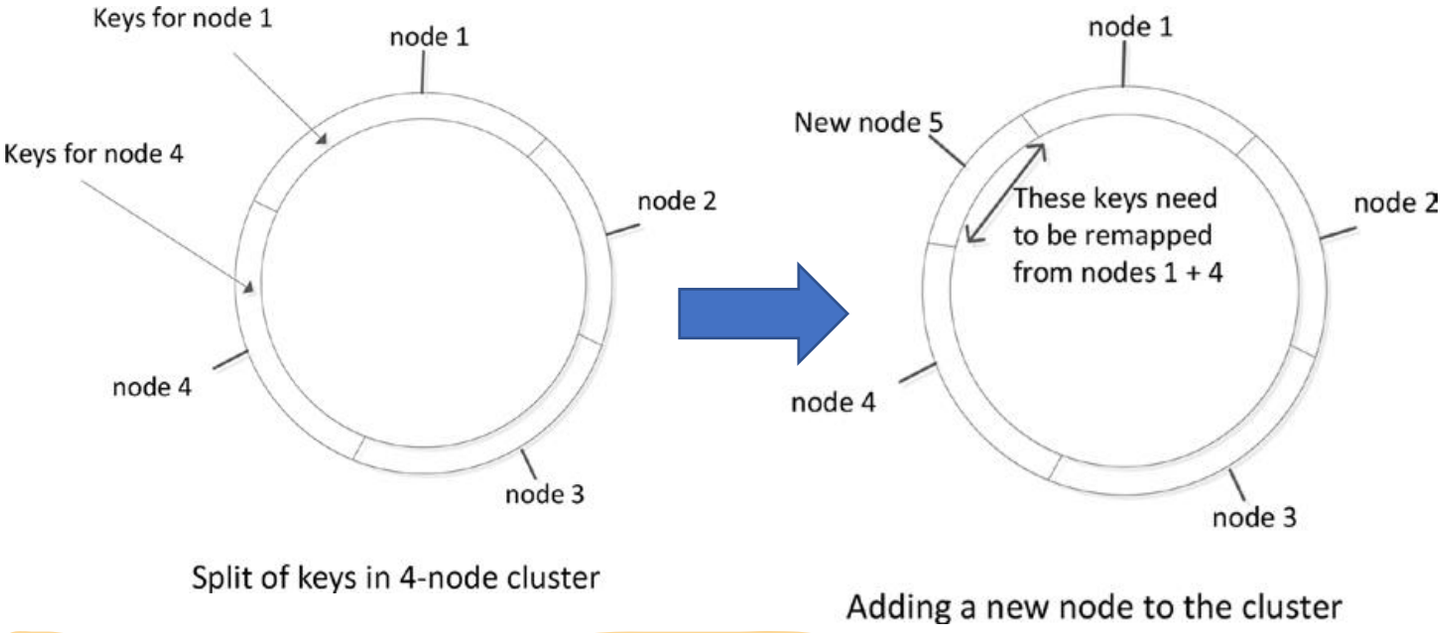
When we hash a key value, we perform a mathematical computation on the key value and use that computed value to determine **where to store the data**. One reason to use hashing is so that we are able to evenly distribute the data across a certain number of slots. The most simple example is to use the modulo function,

which returns the remainder of a division. If we want to hash any number into 10 buckets, we can use modulo 10; then key 27 would map to bucket 7, key 32 would map to bucket 2, key 25 to bucket 5, and so on. Using this method, we could map keys evenly across 10 servers. When we want to determine which node should store a particular item, we would calculate its modulo and use the result to locate the node.

In practice, hashing functions are more complex than a simple modulo function, and a good hash function always distributes the hash values evenly across nodes, regardless of any skew in key values.

Hashing works great as a way of distributing data evenly across a fixed number of nodes. But we have a problem if we add or remove a node—we have to recalculate the hash values and redistribute all the data. For instance, if we wanted to add a new server in the modulo 10 example above, we would recalculate hashes using modulo 11 and then we would have to move almost every data item accordingly. Consistent hashing works by hashing key values and applying a consistent method for allocating those hashed values to specific nodes.

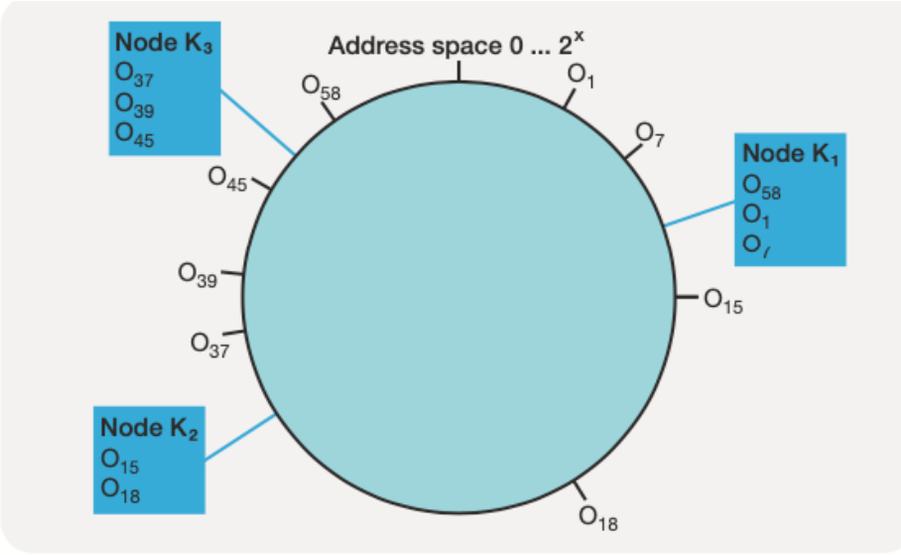
By convention and possibly federal law, consistent hashing schemes are represented as **rings**— because the hash values "loop around" to 0. The figure shows what happens when a node is added to an existing cluster. Only those keys currently mapped to the "neighbors" of the new node need remapping.

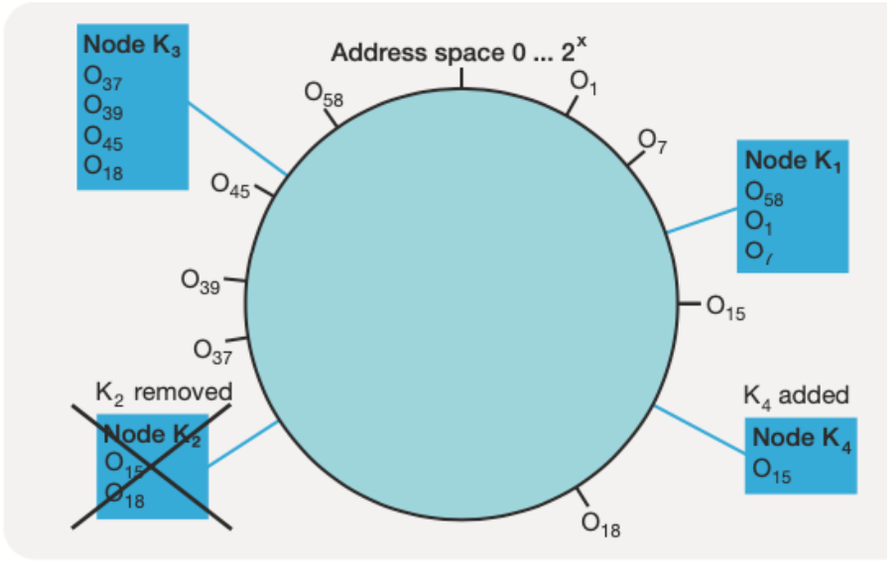


Consistent hashing ensures a good load balance among servers.

The same hashing function is applied to both the keys and the server/partition ID/Address/Name.

The hash values must be in the same range, for example hexadecimal on 32 bit representation





The actual server (Node) kj associated to a specific key (object) oi is its successor in the hashing/address space.

KEY-VALUE IMPLEMENTATION TERMS

HASH FUNCTION

Hash functions are algorithms that map from an input—for example, a string of characters—to an output string. The size of the input can vary, but the size of the output is always the same.

- One of the important characteristics of hash algorithms is that even small changes in the input can lead to large changes in the output.

Hash functions are generally designed to distribute inputs evenly over the set of all possible outputs. The output space can be quite large. The output space can be quite large.

This is especially useful when hashing keys.

The ranges of output values can be assigned to partitions and you can be reasonably assured that each partition will receive approximately the same amount of data. For example, assume you have a cluster of 16

nodes and each node is responsible for one partition. You can use the first digit output by the SHA-1 function to determine which partition should receive the data.

The key 'cust:8983:firstName' has a hash value of

4b2cf78c7ed41fe19625d5f4e5e3eab20b064c24

and would be assigned to partition 4, while the key 'cust:8983:lastName' has a hash value of

c0017bec2624f736b774efdc61c97f79446fc74f and would be assigned to node 12 (c is the hexadecimal digit for the base-10 number 12).

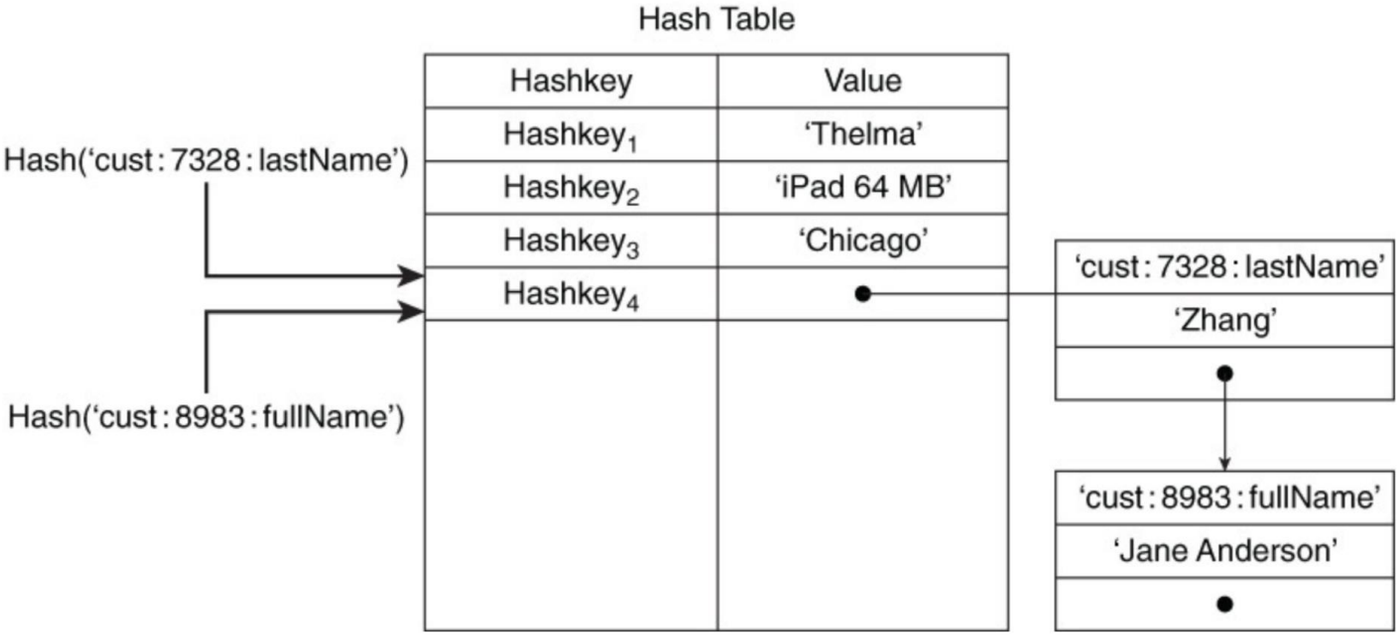
Although there are many possible outputs for hash functions, it is possible for two distinct input strings to map to the same output string.

COLLISION

A collision occurs when two distinct inputs to a hash function produce the same output. When it is difficult to find two inputs that map to the same hash function output, the hash function is known as collision resistant. If a hash table is not collision resistant or if you encounter one of those rare cases in which two inputs map to the same output, you will need a collision resolution strategy.

Basically, a collision resolution strategy is a way to deal with the fact that you have two inputs that map to the same output. If the hash table only has room for one value, then one of the hashed values will be lost.

A simple method to deal with this is to implement a list in each cell of a hash table. Most entries will include a single value, but if there are collisions, the hash table cell will maintain a list of keys and values.



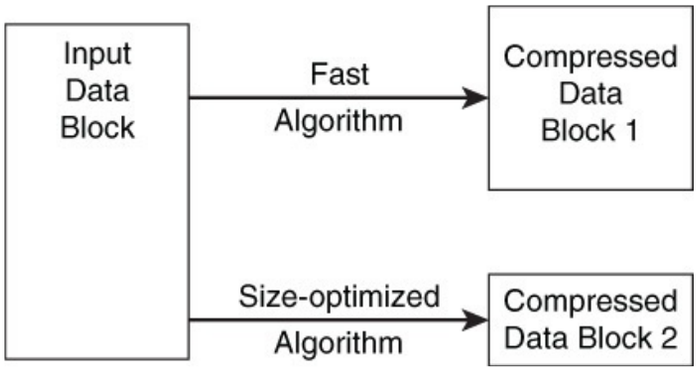
COMPRESSION

Key-value databases are memory intensive. Large numbers of large values can quickly consume substantial amounts of memory. Operating systems can address this problem with virtual memory management, but that entails writing data to disk or flash storage.

Reading from and writing to disk is significantly slower than reading from random access memory, so avoid it when possible. One option is to add more memory to your servers. There are both technical and cost limitations on this option. In the case of disk-based, key-value stores, such as the LevelDB library (code.google.com/p/leveldb/), there is still a motivation to optimize storage because the time required to read and write data is a function of the size of the data.

One way to optimize memory and persistent storage is to use compression techniques. A compression algorithm for key-value stores should perform compression and decompression operations as fast as possible. This often entails a trade-off between the speed of compression/decompression and the size of the compressed data.

Faster compression algorithms can lead to larger compressed data than other, slower algorithms. For example, the Snappy compression algorithm compresses 250MB per second and decompresses 500MB per second on a Core i7, 64-bit mode processor but produces compressed data that is 20% to 100% larger than the same data compressed by other algorithms.4



Look for compression algorithms that ensure a trade-off between the speed of compression/decompression and the size of the compressed data.

USING KEY-VALUE DATABASES

If data organization and management is more important than the performances, classical relational databases are more suitable rather than key-value databases.

However, if we are more interested to the performances (high availability, short response time, etc.) and/or the data model is not too much complicated (no hierarchical organization, limited number of relationships) we may use key-values databases.

Indeed, key-value stores are really simple and easy to handle, data can be modeled in a less complicated manner than in RDBMS.

I database relazionali continuano ad essere necessari in base alle richieste dei requisiti funzionali e non.

Key value database invece sono semplici e veloci, ma solo quando le richieste sono adeguate per questo tipo di datbase.

DOCUMENT DATABASE

A document database is a nonrelational database that stores data as structured documents, usually in XML or **JSON** formats.

Developers often turn to document databases when they need the flexibility of NoSQL databases but need to manage more complex data structures than those readily supported by key-value databases.

Document databases do not require you to define a common structure for all records in the data store (**schema-less**). Document databases, however, do have some similar features to relational databases. For example, it is possible to query and filter collections of documents much as you would rows in a relational table.

Some document databases allows also ACID transactions.

XML DOCUMENTS

XML stands for eXtensible Markup Language. It is a markup language specifies the structure and content of a document.

- Tags are added to the document to provide the extra information.
- XML tags give a reader some idea what some of the data means.
- XML is capable of representing almost any form of information.

XML and Cascading Style Sheets (CSS) allowed second-generation websites to separate data and format.

XML is also the basis for many data interchange protocols and, in particular, was a foundation for web service specifications such as SOAP (Simple Object Access Protocol).

XML is a standard format for many document types, eventually including word processing documents and spreadsheets (docx, xlsx and pptx formats are based on XML).

```
<item>
   <title>Kind of Blue</title>
   <artist>Miles Davis</artist>
   <tracks>
      <track length="9:22">So What</track>
      <track length="9:46">Freddie Freeloader</track>
      <track length="5:37">Blue in Green</track>
      <track length="11:33">All Blues</track>
      <track length="9:26">Flamenco Sketches</track>
   </tracks>
</item>
<item>
   <title>Cookin'</title>
   <artist>Miles Davis</artist>
   <tracks>
      <track length="5:57">My Funny Valentine</track>
      <track length="9:53">Blues by Five</track>
      <track length="4:22">Airegin</track>
      <track length="13:03">Tune-Up</track>
   </tracks>
</item>
<item>
   <title>Blue Train</title>
   <artist>John Coltrane</artist>
   <tracks>
      <track length="10:39">Blue Train</track>
      <track length="9:06">Moment's Notice</track>
      <track length="7:11">Locomotion</track>
      <track length="7:55">I'm Old Fashioned</track>
      <track length="7:03">Lazy Bird</track>
   </tracks>
</item>
```

XML is text (Unicode) based. – Can be transmitted efficiently.

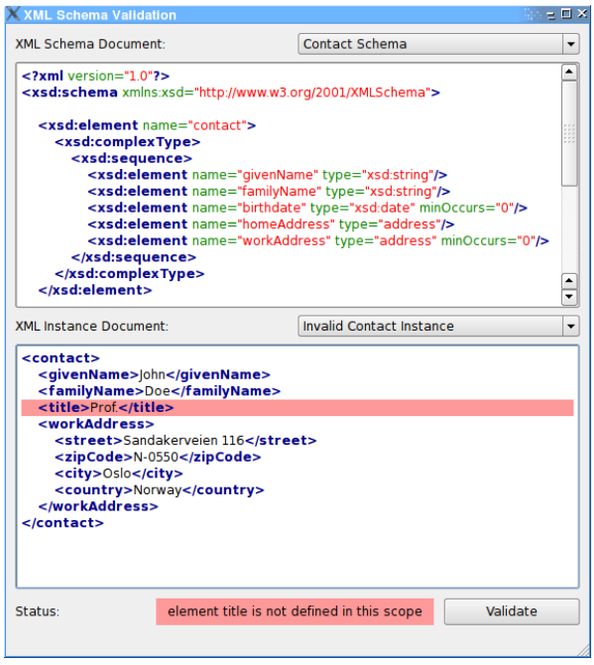
One XML document can be displayed differently in different media and software platforms.

XML documents can be modularized. Parts can be reused.

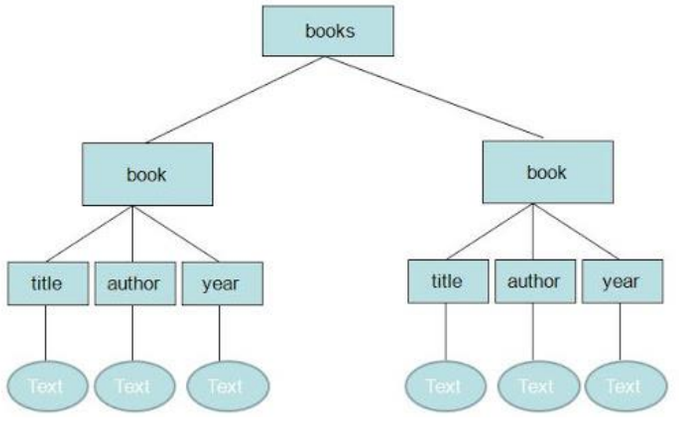
XML ECOSYSTEM

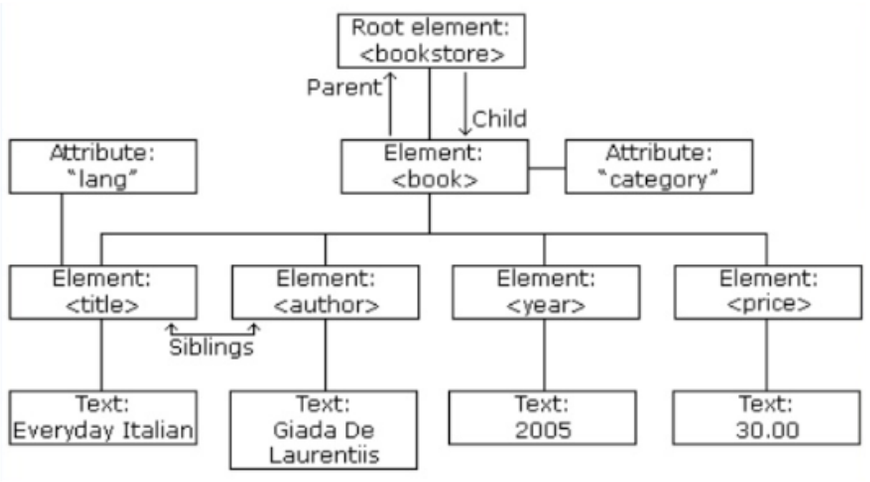
XML is supported by a rich ecosystem that includes a variety of standards and tools to assist with authoring, validation, searching, and transforming XML documents.

- **XPath**: useful to navigate through elements and attributes in an XML document.
- **XQuery**: is the language for querying XML data and is built on XPath expressions.
- **XML schema**: A special type of XML document that describes the elements that may be present in a specified class of XML documents.
- **XSLT** (**Extensible Stylesheet Language Transformations**): A language for transforming XML documents into alternative formats, including non-XML formats such as HTML.
- **DOM**(**Document Object Model**): a platform- and language-neutral interface for dynamically managing the content, structure and style of documents such as XML and XHTML. A document is handled as tree.




```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <bookstore>
      <book category="cooking">
            <title lang="en">Everyday Italian</title>
            <author>Giada De Laurentiis</author>
            <year>2005</year>
            <price>30.00</price>
      </book>
      <book category="web" cover="paperback">
            <title lang="en">Learning XML</title>
            <author>Erik T. Ray</author>
            <year>2003</year>
            <price>39.95</price>
      </book>
  </bookstore>
```





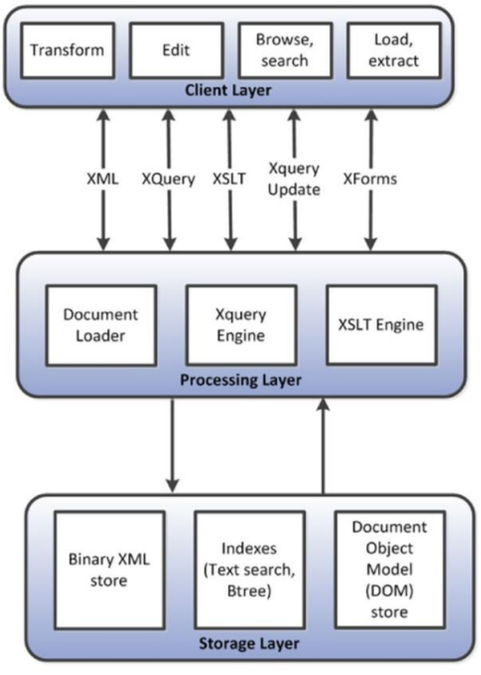
XML DATABASES

XML databases: platforms that implement the various XML standards such as XQuery and XSLT,

They provide services for the storage, indexing, security, and concurrent access of XML files.

XML databases did not represent an alternative for RDBMSs.

On the other hand, some RDBMSs introduced XML , allowing the storage of XML documents within A BLOB (binary large object) columns.



MAIN DRAWBACKS

XML tags are **verbose** and **repetitious**, thus the amount of storage required increases.

XML documents are **wasteful of space** and are also **computationally expensive** to parse.

In general, XML databases are used as **content-management systems**: collections of text files (such as academic papers and business documents) are organized and maintained in XML format.

On the other hand, **JSON-based** document databases are more suitable to support **web-based** operational **workloads**, such as storing and modifying dynamic contents.

JSON DOCUMENTS

JSON acronym of JavaScript Object Notation.

Used to format data.

Thanks to its integration with JavaScript, a JSON document has been often preferred to an XML document for data interchanging on the Internet.

Despite the name, JSON is a (mostly) language-independent way of specifying objects as name-value pairs.

```
{"skillz": {
        "web":[
                {"name": "html",
                 "years": "5"
                },
                {"name": "css",
                 "years": "3"
                }],
        "database":[
                {"name": "sql",
                 "years": "7"
                }]
}}
```

An **object** is an unordered set **of name/value** pairs

- The pairs are enclosed within braces, { }
- There is a colon between the name and the value
- Pairs are separated by commas
    - Example: { "name": "html", "years": 5 }

An **array** is an **ordered collection** of values

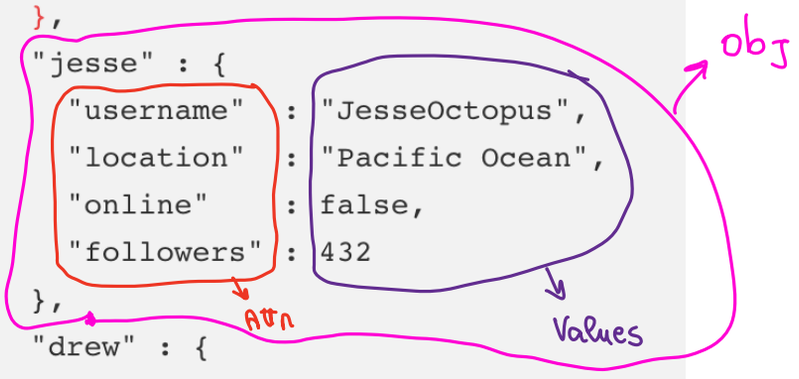
- The values are enclosed within brackets, [ ]
- Values are separated by commas
- Example: ["html", ”xml", "css"]

A value can be

- A string
- a number
- true or false
- null
- an object
- an array

- Values can be nested
- All numbers are decimal--**no octal or hexadecimal**
- Strings are enclosed in double quotes, and can contain the usual assortment of escaped characters
- Numbers have the usual C/C++/Java syntax, including exponential (E) notation

```
{
  "sammy" : {
    "username"  : "SammyShark",
    "location"  : "Indian Ocean",
    "online"    : true,
    "followers" : 987
  },
  "jesse" : {
    "username"  : "JesseOctopus",
    "location"  : "Pacific Ocean",
    "online"    : false,
    "followers" : 432
  },
  "drew" : {
    "username"  : "DrewSquid",
    "location"  : "Atlantic Ocean",
    "online"    : false,
    "followers" : 321
  },
  "jamie" : {
    "username"  : "JamieMantisShrimp",
    "location"  : "Pacific Ocean",
    "online"    : true,
    "followers" : 654
  }
}
```



```
{
  "first_name" : "Sammy",
  "last_name" : "Shark",
  "location" : "Ocean",
  "websites" : [
    {
      "description" : "work",
      "URL" : "https://www.digitalocean.com/"
    },
    {
      "desciption" : "tutorials",
      "URL" : "https://www.digitalocean.com/community/tutorials"
    }
  ],
  "social_media" : [
    {
      "description" : "twitter",
      "link" : "https://twitter.com/digitalocean"
    },
    {
      "description" : "facebook",
      "link" : "https://www.facebook.com/DigitalOceanCloudHosting"
    },
    {
      "description" : "github",
      "link" : "https://github.com/digitalocean"
    }
  ]
}
```

COMPARISON OF JSON AND XML

Similarities

- Both are human readable
- Both have very simple syntax
- Both are hierarchical
- Both are language independent
- Both supported in APIs of many programming languages

Differences

- Syntax is different
- JSON is less verbose
- JSON includes arrays
- Names in JSON must not be JavaScript reserved words

users.xml

```xml
<users>
    <user>
        <username>SammyShark</username> <location>Indian Ocean</location>
    </user>
    <user>
        <username>JesseOctopus</username> <location>Pacific Ocean</location>
    </user>
    <user>
        <username>DrewSquir</username> <location>Atlantic Ocean</location>
    </user>
    <user>
        <username>JamieMantisShrimp</username> <location>Pacific Ocean</location>
    </user>
</users>
```

users.json

```json
{"users": [
  {"username" : "SammyShark", "location" : "Indian Ocean"},
  {"username" : "JesseOctopus", "location" : "Pacific Ocean"},
  {"username" : "DrewSquid", "location" : "Atlantic Ocean"},
  {"username" : "JamieMantisShrimp", "location" : "Pacific Ocean"}
] }
```

MAIN FEATURE OF JSON DATABASES

Data have to be stored in the **JSON format**.

- A **document** is the basic unit of storage, corresponding approximately to a row in an RDBMS. A document comprises one or more key-value pairs, and may also contain nested documents and arrays. Arrays may also contain documents allowing for a complex hierarchical structure.
- A **collection** is a set of documents sharing some common purpose; this is roughly equivalent to a relational table. The documents in a collection don't have to be of the same type, though it is typical for documents in a collection to represent a common category of information.

- Schema less: predefined document elements must not be defined.
- Polymorphic Scheme: the documents in a collection may be different.

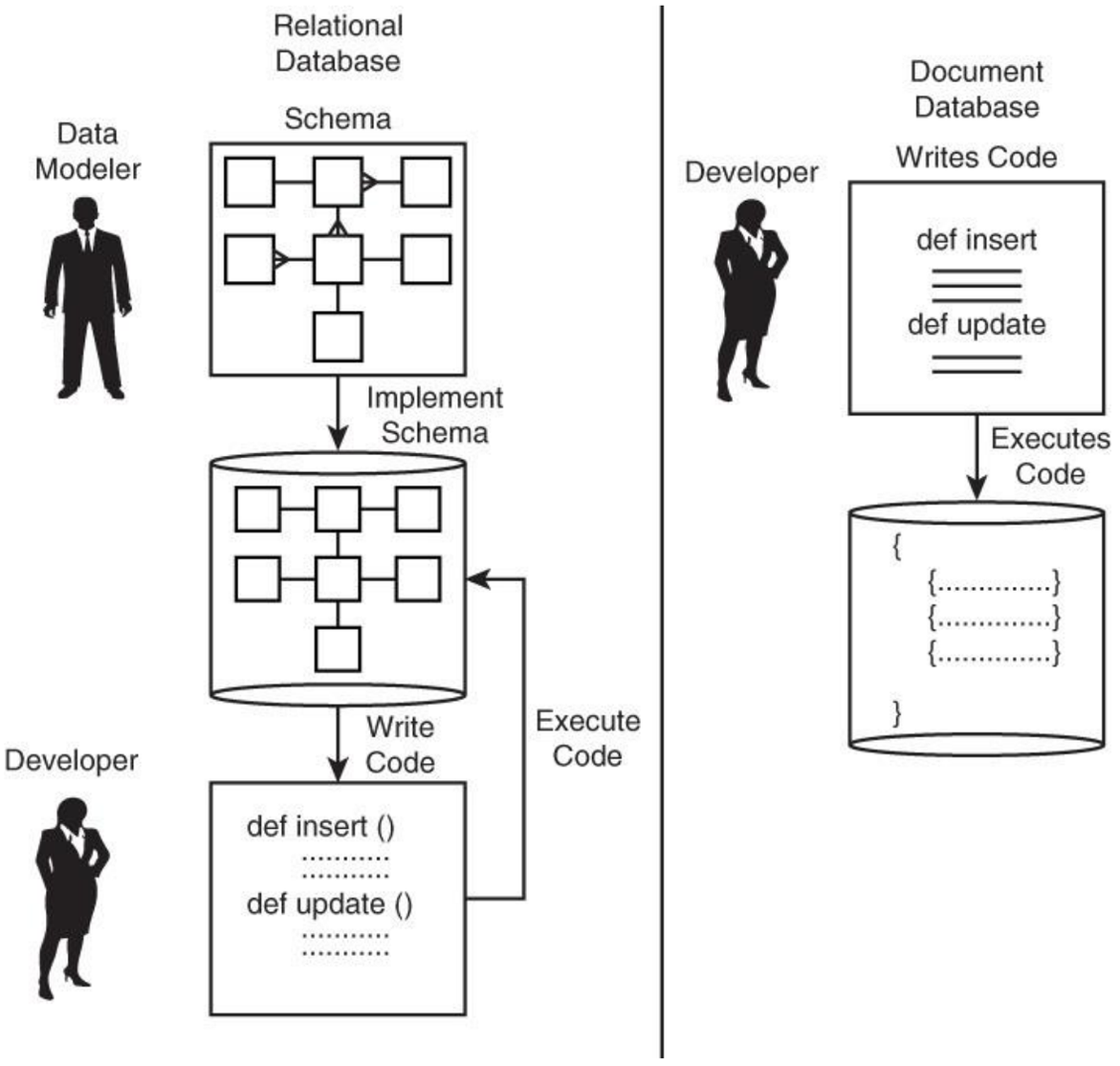
SCHEMA-LESS VS SCHEMA DEFINITION

A schema is a specification that describes the **structure of an object**, such as a table.

Data modelers have to define **tables** in a relational database before **developers** can execute **code** to add, remove, or update rows in the table.

Document databases do not require this formal definition step.

Developers can **create** collections and documents in collections by simply **inserting** them into the database

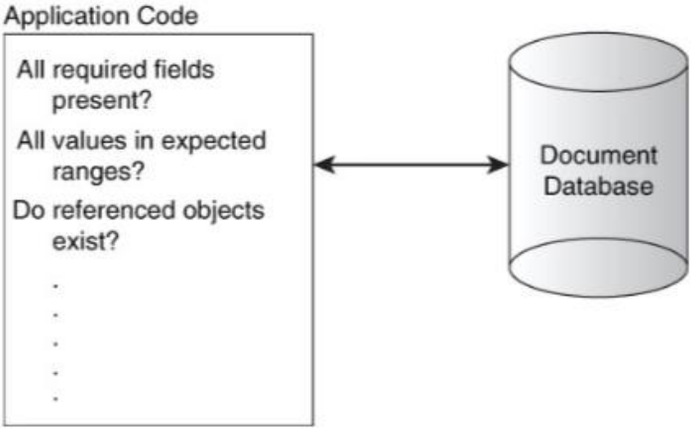


SCHEMA-LESS PROS AND CONS

Pros: High flexibility in handling the structure of the objects to store

```
{  'employeeName' :  'Janice Collins',
   'department' : 'Software engineering'
   'startDate' : '10-Feb-2010',
   'pastProjectCodes' : [ 189847, 187731, 176533, 154812]
}
```

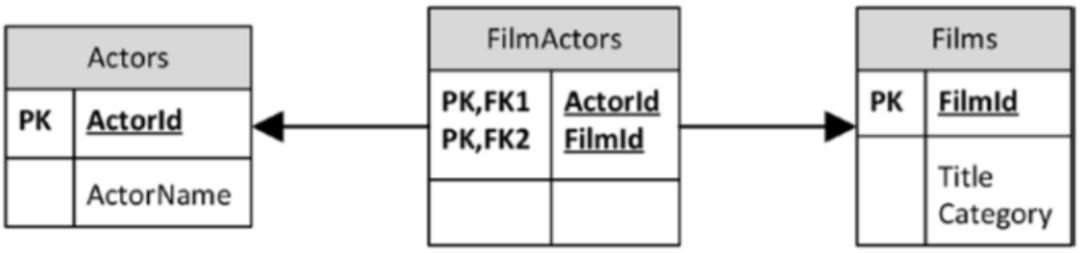
```
{  'employeeName' :  'Robert Lucas,
   'department' : 'Finance'
   'startDate' : '21-May-2009',
   'certifications' : 'CPA'
}
```



Cons: the DBMS may be not allowed to enforce rules based on the structure of the data.

CONSIDERATIONS

A document database could theoretically implement a third normal form schema. Tables, as in relational databases, may be “simulated” considering collections with JSON documents with an identical pre-defined structure.



JSON DATABASES: AN EXAMPLE

Document databases usually adopts a **reduced number** of collections for modeling data.

- **Nested documents** are used for representing **relationships** among the different entities.
- Document databases **do not** generally provide **join operations**.
- **Programmers like** to have the JSON structure map **closely to the object** structure of their code!!!

```
{ "_id" : 97, "Title" : "BRIDE INTRIGUE",
              "Category" : "Action",
    "Actors" :
      [ { "actorId" : 65, "Name" : "ANGELA HUDSON" } ]
}

{ "_id": 115,"Title": "CAMPUS REMEMBER",
             "Category": "Action",
    "Actors" :
       [
         { "actorId": 8,"Name": "MATTHEW JOHANSSON" },
         { "actorId": 45,"Name": "REESE KILMER" },
         { "actorId": 168,"Name": "WILL WILSON" }
       ]
}

{ "_id" : 105, "Title" : "BULL SHAWSHANK",
               "Category" : "Action",
     "Actors" :
       [ { "actorId" : 2, "Name" : "NICK WAHLBERG" },
         { "actorId" : 23, "Name" : "SANDRA KILMER" } ]
}
```

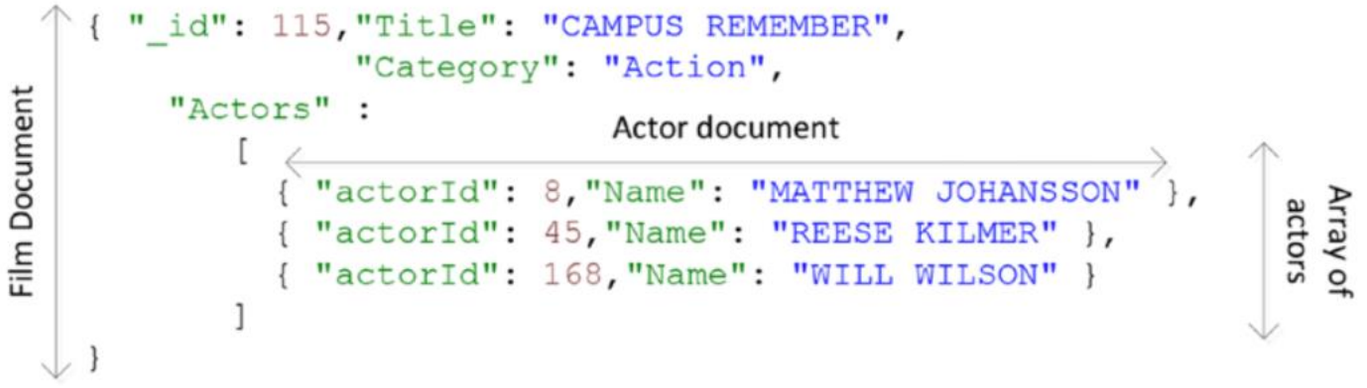
Actor document

Film Document

Array of actors

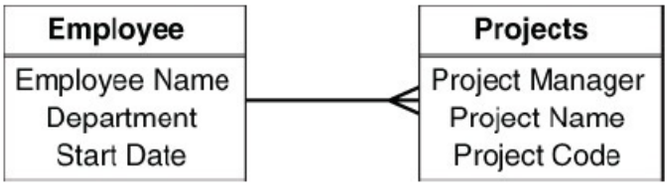
Collection

DATA MODELING: DOCUMENT EMBEDDING

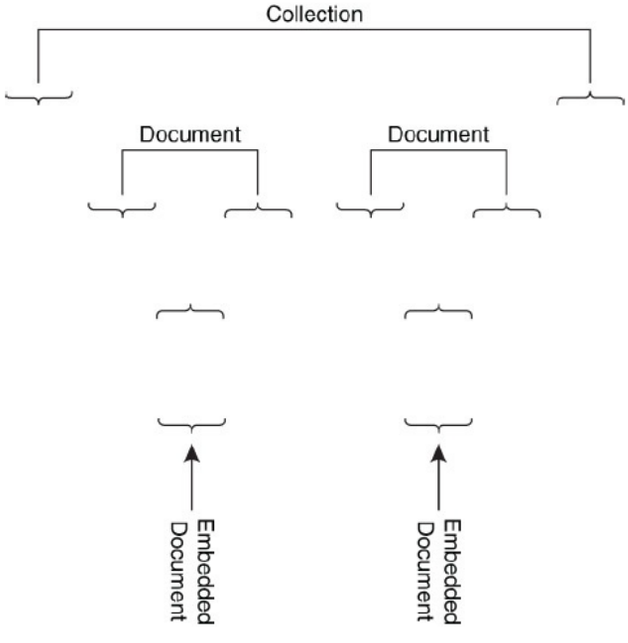


- The solution above allows the user to **retrieve** a film and all its actors in a **single operation**.
- However, "actors" result to be **duplicated** across multiple documents.
- In a complex design this could lead to **issues** and possibly **inconsistencies** if any of the "actor" attributes need to be changed.
- Moreover, some JSON databases have some **limitations** of the maximum **dimension** of a single document.

One of the advantages of document databases is that they allow developers to store related data in more flexible ways than typically done in relational databases. If you were to model employees and the projects they work on in a relational database, you would probably create two tables: one for employee information and one for project information:



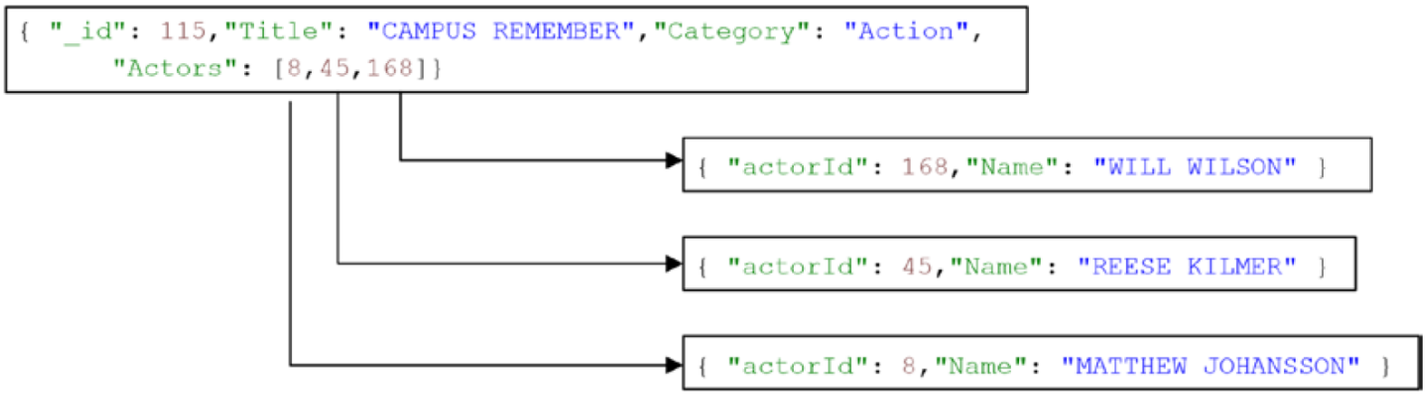
An embedded document enables document database users to store related data in a single document. This allows the document database to avoid a process called joining in which data from one table, called the foreign key, is used to look up data in another table.



Joining two large tables can be potentially time consuming and require a significant number of read operations from disk. Embedded documents allow related data to be stored together. When the document is read from disk, both the primary and the related information are read without the need for a join operation.

- Embedded documents are documents within documents. They are used to improve database performance by storing together data that is frequently used together.

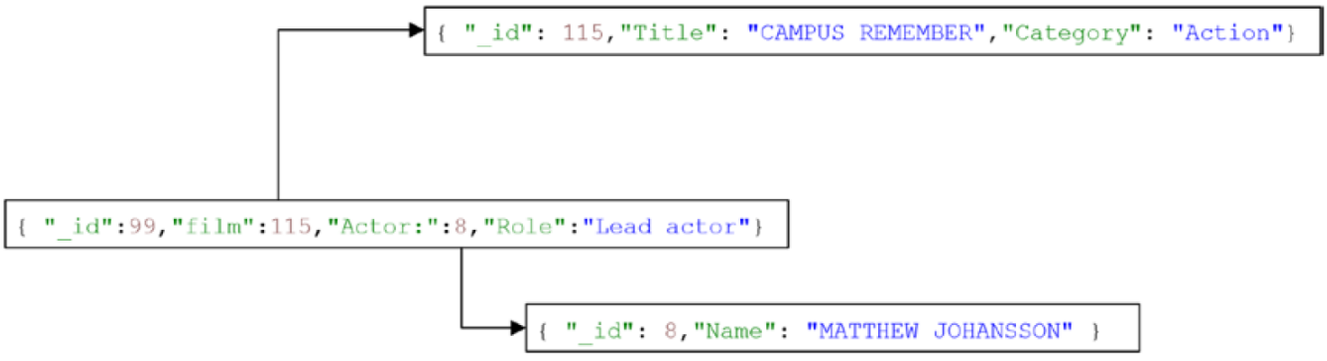
DATA MODELING: DOCUMENT LINKING



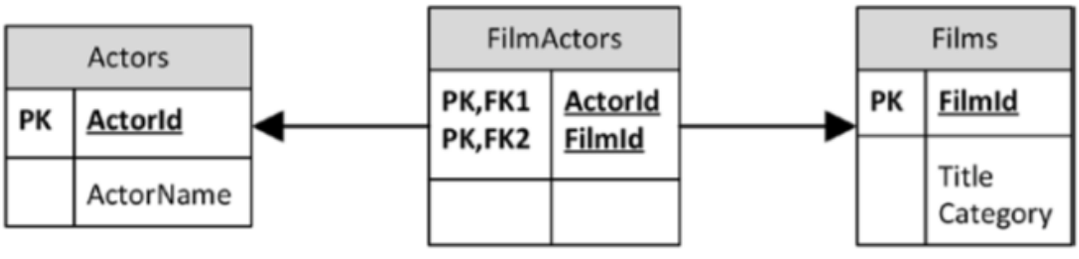
In the solution above, an array of actor **IDs** has been embedded into the film document.

The IDs can be used to **retrieve** the documents of the actors (in on other collection) who appear in a film.

We are rolling back to a relational model!!! Now, at least two collections of document must be defined.



We are rolling back to a the third normal form!!!!!



Document linking approaches are usually somewhat an **unnatural** style for a document database.

However, for some workloads it may provide the best balance between performance and maintainability.

When modeling data for document databases, there is no equivalent of third normal form that defines a "correct" model.

In this context, the nature of the queries to be executed drives the approach to model data.

DATA MODELING: ONE TO MANY RELATIONSHIP

Let consider the customer entity which may have associated a list of address entities.

```
{
   customer_id: 76123,
   name: 'Acme Data Modeling Services',
   person_or_business: 'business',
   address : [
                    { street: '276 North Amber St',
                       city: 'Vancouver',
                       state: 'WA',
                       zip: 99076} ,
                    { street: '89 Morton St',
                       city: 'Salem',
                       state: 'NH',
                       zip: 01097}
                   ]
   }
```

The basic pattern is that the **one** entity in a one-to-many relation is the **primary document**, and the many entities are represented as an array of **embedded documents**.

DATA MODELLING: MANY TO MANY RELATIONSHIPS

Let consider an example of application in which:

- A student can be enrolled in many courses
- A course can have many students enrolled to it

We can model this situation considering the following two collections:

```
{
  { courseID: 'C1667',
     title: 'Introduction to Anthropology',
     instructor: 'Dr. Margret Austin',
     credits: 3,
     enrolledStudents: ['S1837', 'S3737', 'S9825' …
       'S1847'] },
  { courseID: 'C2873',
     title: 'Algorithms and Data Structures',
     instructor: 'Dr. Susan Johnson',
     credits: 3,
     enrolledStudents: ['S1837','S3737', 'S4321', 'S9825'
       … 'S1847'] },
  { courseID: C3876,
     title: 'Macroeconomics',
     instructor: 'Dr. James Schulen',
     credits: 3,
     enrolledStudents: ['S1837', 'S4321', 'S1470', 'S9825'
       … 'S1847'] },
```

```
{
 {studentID:'S1837',
   name: 'Brian Nelson',
   gradYear: 2018,
   courses: ['C1667', C2873,'C3876']},
 {studentID: 'S3737',
   name: 'Yolanda Deltor',
        gradYear: 2017,
        courses: [ 'C1667','C2873']},
    …
}
```

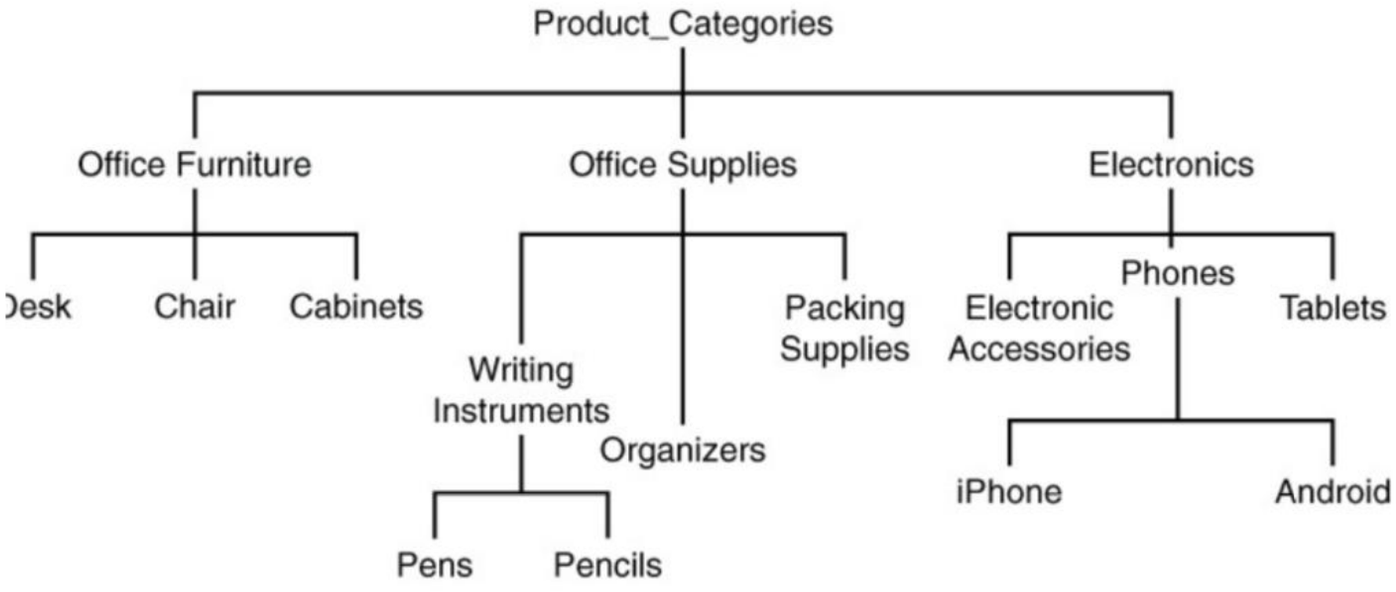
The pattern minimizes duplicate data by referencing related documents with identifiers instead of embedded documents.

We have to take care when updating data in this kind of relationship. Indeed, the DBMS will not control the referential integrity as in relational DBMSs.

Care must be taken when updating many-to-many relationships so that both entities are correctly updated. Also remember that document databases will not catch referential integrity errors as a relational database will. Document databases will allow you to insert a student document with a courseID that does not correspond to an existing course.

MODELING HIERARCHIES

Hierarchies describe instances of entities in some kind of parent-child or part-subpart relation.



There are a few different ways to model hierarchical relations. Each works well with particular types of queries.

1) PARENT REFERENCES

A simple technique is to keep a reference to either the parent or the children of an entity (create a collection for each branch). Could model product categories with references to their parents:

parent reference

```
{
   {productCategoryID: 'PC233', name:'Pencils',
     parentID:'PC72'},
   {productCategoryID: 'PC72', name:'Writing Instruments',
     parentID: 'PC37"},
   {productCategoryID: 'PC37', name:'Office Supplies',
     parentID: 'P01'},
   {productCategoryID: 'P01', name:'Product Categories' }
}
```

Notice that the root of the hierarchy, 'Product Categories', does not have a parent and so has no parent field in its document.

This solution is useful if we have frequently to show a specific instance of an object and then show the more general type of that category.

2) CHILD REFERENCE

Child Reference

```
{
   {productCategoryID: 'P01', name:'Product Categories',
         childrenIDs: ['P37','P39','P41']},
        {productCategoryID: 'PC37', name:'Office Supplies',
          childrenIDs: ['PC72','PC73','PC74"]},
         {productCategoryID: 'PC72', name:'Writing
           Instruments', childrenIDs: ['PC233','PC234']'},
          {productCategoryID: 'PC233', name:'Pencils'}
   }
```

The bottom nodes of the hierarchy, such as 'Pencils', do not have children and therefore do not have a childrenIDs field.

This pattern is useful if you routinely need to retrieve the children or subparts of the instance modeled in the document. For example, if you had to support a user interface that allowed users to drill down, you could use this pattern to fetch all the children or subparts of the current level of the hierarchy displayed in the interface.

3) LIST OF ANCESTOR

```
{productCategoryID: 'PC233', name:'Pencils',
  ancestors:['PC72', 'PC37', 'P01']}
```

↳ Antenati

This pattern is useful when you have to know the full path from any point in the hierarchy back to the root.

An advantage of this pattern is that you can retrieve the full path to the root in a single read operation. Using a parent or child reference requires multiple reads, one for each additional level of the hierarchy.

A disadvantage of this approach is that changes to the hierarchy may require many write operations. The higher up in the hierarchy the change is, the more documents will have to be updated. For example, if a new level was introduced between 'Product Category' and 'Office Supplies', all documents below the new entry would have to be updated. If you added a new level to the bottom of the hierarchy—for example, below 'Pencils' you add 'Mechanical Pencils' and 'Non-mechanical Pencils'—then no existing documents would have to change.

AN EXAMPLE – DOCUMENT GROWTH

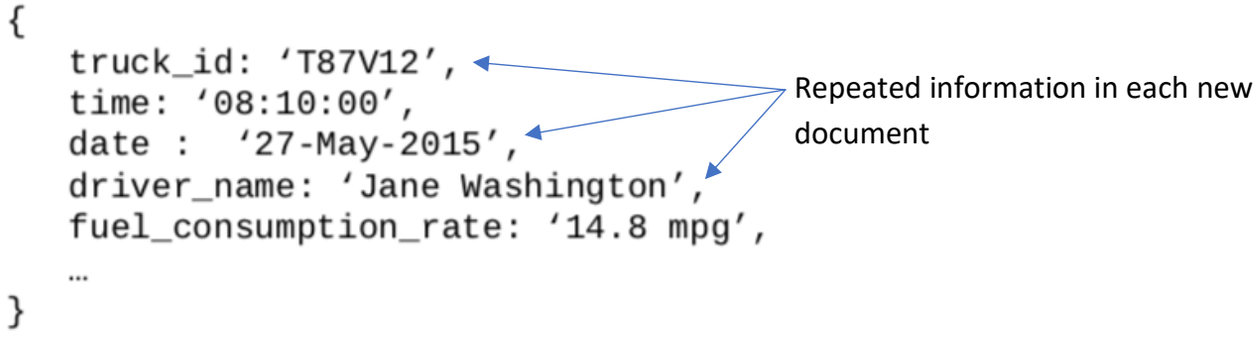
Some documents will **change frequently**, and others will change infrequently. A document that keeps a counter of the number of times a web page is viewed could change hundreds of times per minute. A table that stores server event log data may only change when there is an error in the load process that copies event data from a server to the document database. When designing a document database, consider not just how frequently a document will change, but also how the size of the document may change.

Incrementing a counter or correcting an error in a field will not significantly change the size of a document.

Scenario: Trucks in a company fleet have to transmit location, fuel consumption and other metrics every three minutes to a fleet management data base (one-to-many relationship between a truck and the transmitted details).

We may consider to generate a new document to add to the DB for each data transmission.

Over time, the number of data sets written to the database increases. How should an application designer structure the documents to handle such input streams? One option is to create a new document for each new set of data. In the case of the trucks transmitting operational data, this would include a truck ID, time, location data, and so on:



At the end of the day, the DB will include 200 new documents for each truck (we consider 20 transmissions per hour, 10 working hours)

Each truck would transmit 20 data sets per hour, or assuming a 10-hour operations day, 200 data sets per day. The truck_id, date, and driver_name would be the same for all 200 documents. This looks like an obvious candidate for embedding a document with the operational data in a document about the truck used on a particular day. This could be done with an array holding the operational data documents:

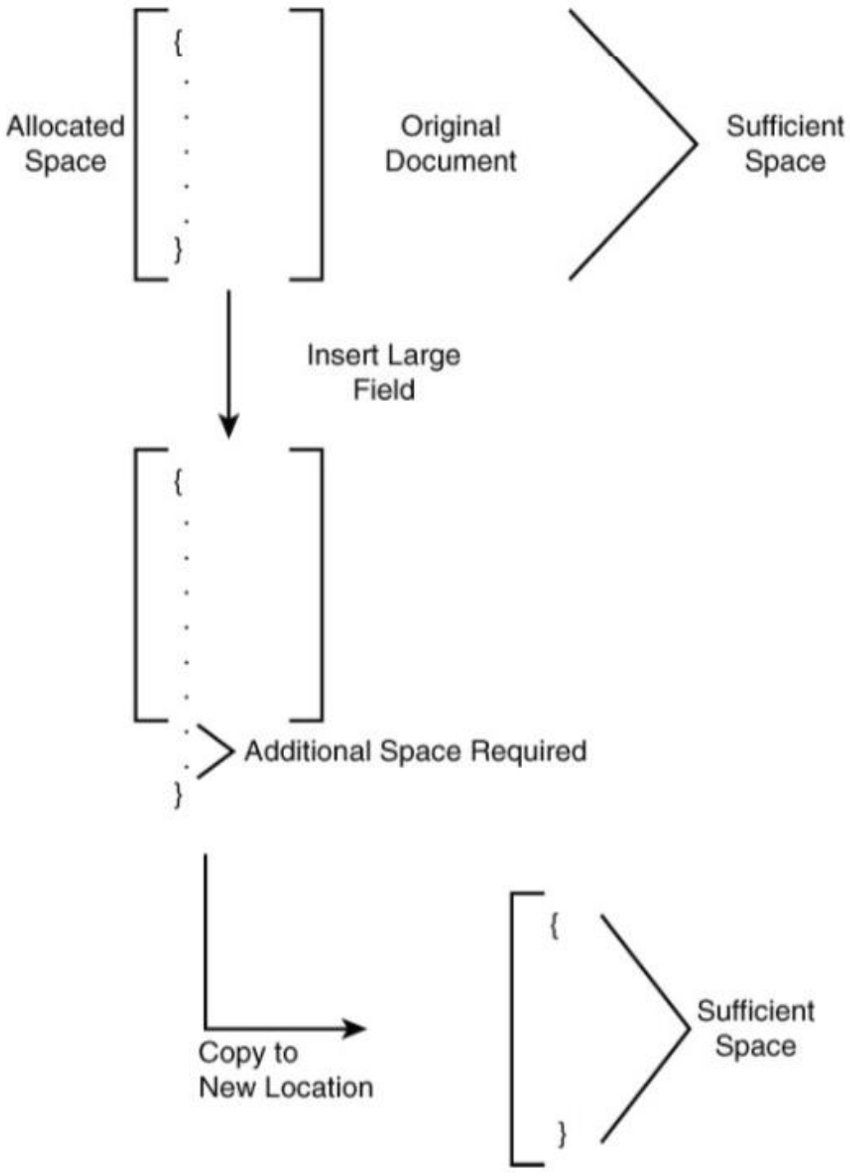
```
{
   truck_id: 'T87V12',
   date :  '27-May-2015',
   driver_name: 'Jane Washington',
   operational_data:
              [
                 {time : '00:01',
                  fuel_consumption_rate: '14.8 mpg',
                  …},
                   {time : '00:04',
                  fuel_consumption_rate: '12.2 mpg',
                  …},
                   {time : '00:07',
                  fuel_consumption_rate: '15.1 mpg',
                  …},
               …]
}
```

Pay attention: we can have a potential performance problem!

The document would start with a single operational record in the array, and at the end of the 10-hour shift, it would have 200 entries in the array.

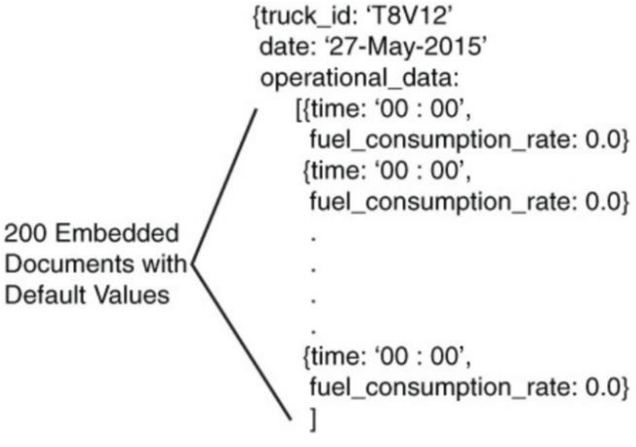
From a logical modeling perspective, this is a perfectly fine way to structure the document, assuming this approach fits your query requirements. From a physical model perspective, however, there is a potential performance problem.

When a document is created, the database management system allocates a certain amount of space for the document. This is usually enough to fit the document as it exists plus some room for growth. If the document **grows** larger than the size allocated for it, the document may be moved to another location. This will require the database management system to read the existing document and copy it to another location, and **free** the previously used storage space



The previous steps can adversely affect the system performance.

AVOID MOVING OVERSIZED DOCUMENTS



One way to avoid this problem of moving oversized documents is to allocate sufficient space for the document at the time the document is created. In the case of the truck operations document, you could create the document with an array of 200 embedded documents with the time and other fields specified with default values. When the actual data is transmitted to the database, the corresponding array entry is updated with the actual values

Consider the life cycle of a document and when possible plan for anticipated growth. Creating a document with sufficient space for the full life of the document can help to avoid I/O overhead.

INDEXING DOCUMENT DATABASE

In order to avoid the entire scan of the overall database, DBMSs for document databases (for example MongoDB) allow the definition of **indexes**.

Indexes, like in book indexes, are a **structured set of information** that maps from one attribute to related information.

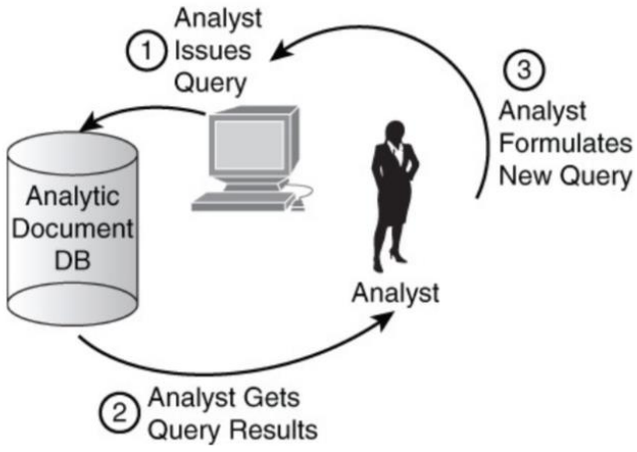
In general, indexes are **special data structures** that store a small portion of the collection's data set in an **easy to traverse** form.

The index stores the value of a specific field or set of fields, **ordered by the value of the field**.

The ordering of the index entries supports **efficient equality matches** and **range-based query operations**.

READ-HEAVY APPLICATIONS

Some applications have a **high percentage of read operations** relative to the number of write operations. Business intelligence and other analytic applications can fall into this category. Read-heavy applications should have indexes on virtually all fields used to help filter results. For example, if it was common for users to query documents from a particular sales region or with order items in a certain product category, then the sales region and product category fields should be indexed.



It is sometimes difficult to know which fields will be used to filter results. This can occur in business intelligence applications. An analyst may explore data sets and choose a variety of different fields as filters. Each time he runs a new query, he may learn something new that leads him to issue another query with a different set of filter fields. This iterative process can continue as long as the analyst gains insight from queries.

Read-heavy applications can **have a large number of indexes**, especially when the query patterns are unknown. It is not unusual to index most fields that could be used to filter results in an analytic application

WRITE-HEAVY APPLICATIONS

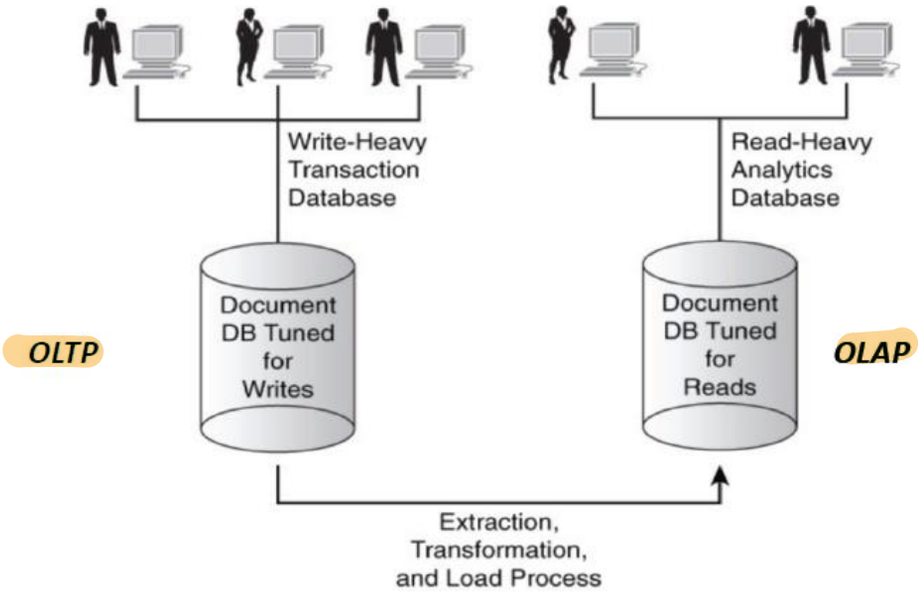
Write-heavy applications are those with **relatively high percentages of write operations** relative to read operations. The document database that receives the truck sensor data described previously would likely be a write-heavy database. Because indexes are data structures that must be created and updated, their use will consume CPU, persistent storage, and memory resources and increase the time needed to insert or update a document in the database.

Data modelers tend to try to **minimize the number of indexes in write-heavy applications**. Essential indexes, such as those created for fields storing the identifiers of related documents, should be in place. As with other

design choices, deciding on the number of indexes in a write-heavy application is a matter of **balancing competing interests**.

Fewer indexes typically correlate with faster updates but potentially slower reads. If users performing read operations can tolerate some delay in receiving results, then minimizing indexes should be considered. If, however, it is important for users to have low-latency queries against a write-heavy database, consider implementing a second database that aggregates the data according to the time-intensive read queries. This is the basic model used in business intelligence.

Transaction processing systems are designed for fast writes and targeted reads. Data is copied from that database using an extraction, transformation, and load (ETL) process and placed in a data mart or data warehouse. The latter two types of databases are usually heavily indexed to improve query response time.



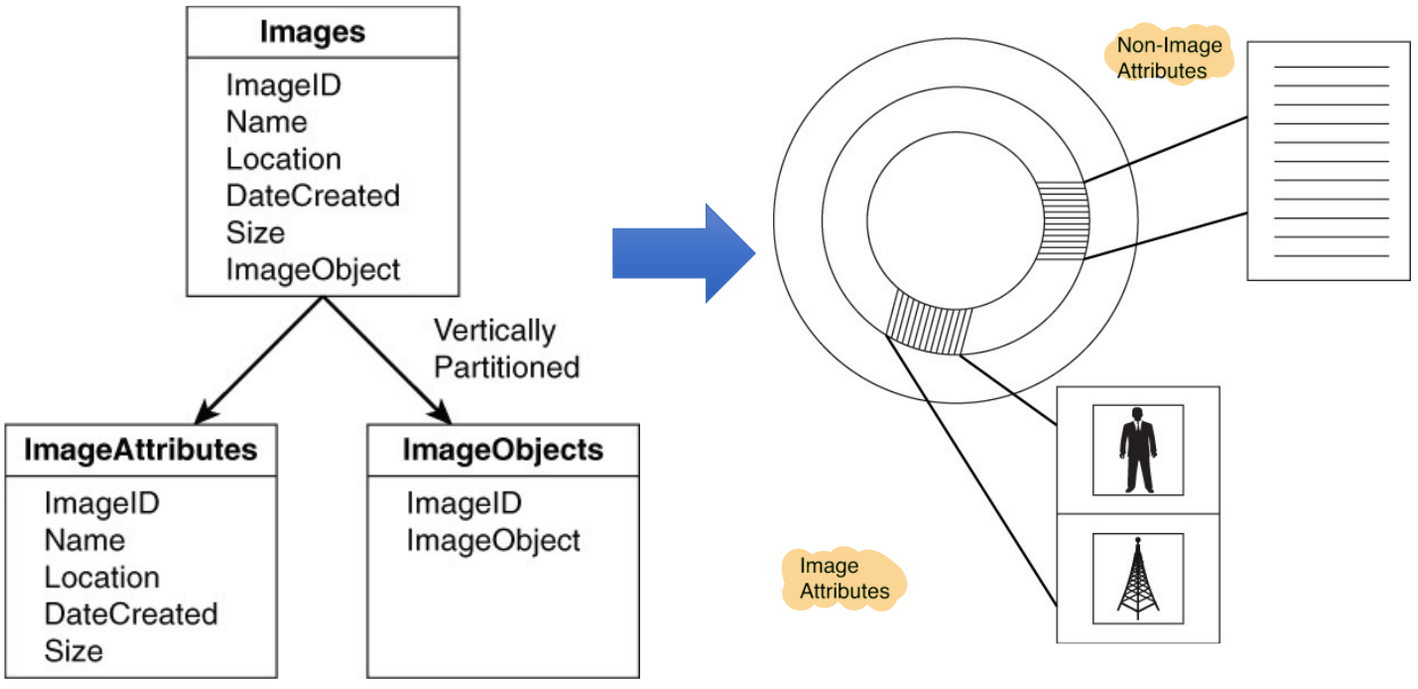
During the late 1980s and 1990s, an increasing number of relational databases were tasked with supporting analytic and decision support applications that often demanded interactive response times. These systems became known as data warehouses, and increasingly were operated parallel to the OLTP system that had generated the original data. The acronym OLAP (Online Analytic Processing) was coined to differentiate these workloads from those of OLTP systems.

Separating OLTP and OLAP workloads was important for maintaining service-level response times for the OLTP systems: sudden IO intensive aggregate queries would generally cause an unacceptable response-time degradation in the OLTP system. But equally important, the OLAP system demanded a different schema from the OLTP system. Star schemas were developed to create data warehouses in which aggregate queries could execute quickly and which would provide a predictable schema for Business Intelligence (BI) tools. In a star schema, central large fact tables are associated with numerous smaller dimension tables. When the dimension tables implement a more complex set of foreign key relationships, then the schema is referred to as a snowflake schema.

(Riassumere il paragrafo sopra, preso dall'altro libro – column database).

VERTICAL PARTITIONING

Vertical partitioning is a technique for improving database performance by **separating columns of a relational table into multiple separate tables**.



This technique is particularly useful when you have some columns that are frequently accessed and others that are not. Consider a table with images and attributes about those images, such as name, location, date image created, and so on. The table of images may be used in an application that allows users to look up images by characteristics.

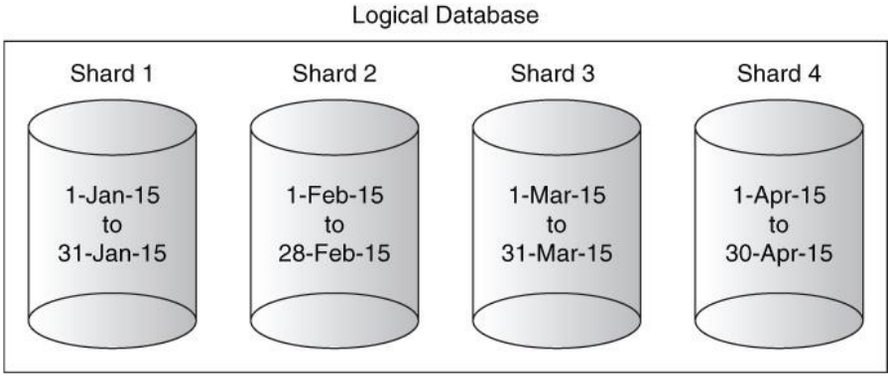
Someone might want pictures from Paris, France, taken within the last three months. The database management system would probably use an index to find the rows of the table that meet the search criteria. If the application only lists the attributes in the resultset and waits for a user to pick a particular record before showing the image, then there is no reason to retrieve the image from the database along with the attributes.

If the image attributes and the image object were stored in the same table, reading the attributes could also force a reading of the image because of layout of data on the disk. By separating the image table into a table of image attributes and the image object, the database can more efficiently retrieve data for the application.

Vertical partitioning is more frequently used with relational database management systems than with document management systems. There are methods for implementing vertical partitioning in nonrelational databases, but horizontal partitioning, or sharding, is more common.

HORIZONTAL PARTITIONING OR SHARDING

Horizontal partitioning is the process of **dividing data into blocks or chunks**. These parts of the database, known as shards, are stored on a specific node (server) of a cluster. Each node can contain only one shard. A single shard may be stored on multiple servers when a database is configured to replicate data. If data is replicated or not, a server within a document database cluster will have only one shard per server.



- Advantages of sharding:
  - Allows handling **heavy loads** (to deploy a larger server with more CPU cores, more memory, and more bandwidth) and the **increase** of system users.
  - Data may be **easily distributed** on a variable number of servers that may be **added** or **removed** by request.
  - Cheaper than vertical scaling (adding ram and disks, upgrading CPUs to a single server).
  - Combined with **replications**, ensures a **high availability** of the system and **fast** responses
    - additional servers can be added to a cluster as demand for a document database grows. Existing servers are not replaced but continue to be used.
  - Combined with replications, ensures a high availability of the system and fast responses.

To implement sharding, document database designers have to select a shard key and a partitioning method.

SHARD KEYS

A shard key is one or more keys or fields that exist in all documents in a collection that is used to separate documents.

Examples of shard keys may be: Unique document ID, Name, Date, such as creation date, Category or type, Geographical region.

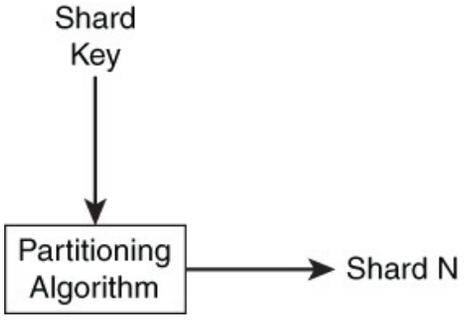
Actually, any atomic field in a document may be chosen as a shard key.

In spite of the discussion that document databases are schemaless, some elements of document databases parallel the schema of relational databases. The use of indexes is one such parallel. Indexes are part of the physical data model of a relational database, which means there is a data structure in the database that implements the index. Indexes are part of the schema of relational databases. Schemaless databases, such as document databases, can have schemalike objects such as indexes as well.
Indexes help improve the speed of read operations and are useful when implementing sharding. Because all documents in a collection need to be placed into a shard, it is important for all documents to have the shard key.

PARTITION ALGORITHMS

There are three main categories of partition algorithms, based on:

- **Range** partition: when you have an ordered set of values for shard keys, such as dates and numbers.
    - For example, if all documents in a collection had a creation date field, it could be used to partition documents into monthly shards. Documents created between January 1, 2015, and January 31, 2015, would go into one shard, whereas documents created between February 1, 2015, and February 28, 2015, would go into another.
- **Hash** partition: uses a hash function to determine where to place a document. Hash functions are designed to generate values evenly across the range of values of the hash function.
    - For example, you have an eight-server cluster, and your hash function generated values between 1 and 8, you should have roughly equal numbers of documents placed on all eight servers.
- **List-based** partitioning: uses a set of values to determine where to place data. You can imagine a product database with several types, including electronics, appliances, household goods, books, and clothes. These product types could be used as a shard key to allocate documents across five different servers.



TIPS ON DESIGNING COLLECTIONS

Collections are **sets** of documents. Because collections do not impose a consistent structure on documents in the collection, it is possible to store **different types** of documents in a single collection. You could, for example, store customer, web clickstream data, and server log data in the same collection. In practice, this is not advisable.

In general, collections **should store** documents about the same type of entity. The concept of an entity is fairly abstract and leaves a lot of room for interpretation. You might consider both web clickstream data and server log data as a "system event" entity and, therefore, they should be stored in the same collection.

```
{ "id" : 12334578,
    "datetime" :  "201409182210",
    "session_num" : 987943,
    "client_IP_addr" : "192.168.10.10",
    "user_agent" : "Mozilla / 5.0",
    "referring_page" : "http://www.example.com/page1"
}
```

Web clickstream data

```
{ “id” : 31244578,
    “datetime” :  “201409172140”,
    “event_type” : “add_user”,
    “server_IP_addr” : “192.168.11.11”,
    “descr” :  “User jones added with sudo privileges”
}
```

Server log data

AVOID HIGHLY ABSTRACT ENTITY TYPES

A system event entity such as this is probably too abstract for practical modeling. This is because web clickstream data and server log data will have few common fields. They may share an ID field and a time stamp but few other attributes. The web clickstream data will have fields capturing information about web pages, users, and transitions from one page to another. The server log documents will contain details about the server, event types, severity levels, and perhaps some descriptive text. Notice how dissimilar web clickstream data is from server log data:

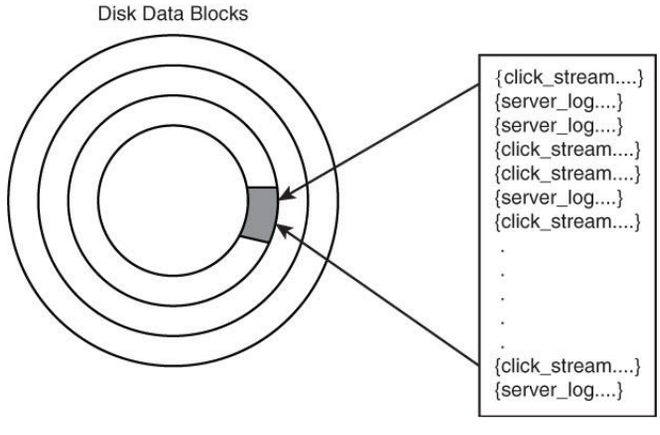
```
{ "id" : 12334578,
    "datetime" :  "201409182210",
    "doc_type": "click_stream",
    "session_num" : 987943,
    "client_IP_addr" : "192.168.10.10",
    "user_agent" : "Mozilla / 5.0",
    "referring_page" : "http://www.example.com/page1"
}

{ "id" : 31244578,
    "datetime" :  "201409172140"
    "doc_type" : "server_log"
    "event_type" : "add_user"
    "server_IP_addr" : "192.168.11.11"
    "descr" :  "User jones added with sudo privileges"
}
```

If you were to store these two document types in a single collection, you would likely need to add a type indicator so your code could easily distinguish between a web clickstream document and a server log event document.

Filtering collections is often slower than working directly with multiple collections, each of which contains a single document type. Consider if you had a system event collection with 1 million documents: 650,000 clickstream documents and 350,000 server log events. Because both types of events are added over time, the document collection will likely store a mix of clickstream and server log documents in close proximity to each other.

If you are using disk storage, you will likely retrieve blocks of data that contain both clickstream and server log documents. This will adversely impact performance.

Mixing document types can lead to **multiple document types** in a disk data **block**. This can lead to inefficiencies because data is read from disk but not used by the application that filters documents based on type.



If the decide to use indexes, recall that they reference a data block that contains both clickstream and server log

data, the disk will read both types of records even though one will be filtered out in your application.

In the case of large collection and large number of distinct “types” of documents, it may be faster to scan the full document collection rather than use an index .

Store different types of documents in the same collection only if they will be frequently used together in the application.

SEPARATE FUNCTIONS FOR MANIPULATING DIFFERENT DOCUMENT TYPES

In general, the application code written for manipulating a collection should have:

1) A substantial amounts of code that apply to all documents

2) Some amount of code that accommodates specialized fields in some documents.

Collection should be broken into multiple collections.

For example, most of the code you would write to insert, update, and delete documents in the customer collection would apply to all documents. You would probably have additional code to handle loyalty and discount fields that would apply to only a subset of all documents.

High-Level Branching

```
   doc.
If (doc_type = ‘click_stream’):
     process_click_stream (doc)
Else
     process_server_log (doc)
```

Lower-Level Branching

```
book.title = doc.title
book.author = doc.author
book.year = doc.publication_year
book.publisher = doc.publisher
book.descr = book.title + book.author + book.year + book.publisher
if (doc.ebook = true);
  book.descr = book.descr + doc.ebook_size
```

High-level branching in functions manipulating documents can indicate a need to create separate collections. Branching at lower levels is common when some documents have optional attributes.

If your code at the highest levels consists of if statements conditionally checking document types that branch to separate functions to manipulate separate document types, it is a good indication you probably have mixed document types that should go in separate collections

DOCUMENT SUBTYPES WHEN ENTITIES ARE FREQUENTLY AGGREGATED OR SHARE SUBSTANTIAL CODE

There are times when it makes sense to use document type indicators and have separate code to handle the different types.

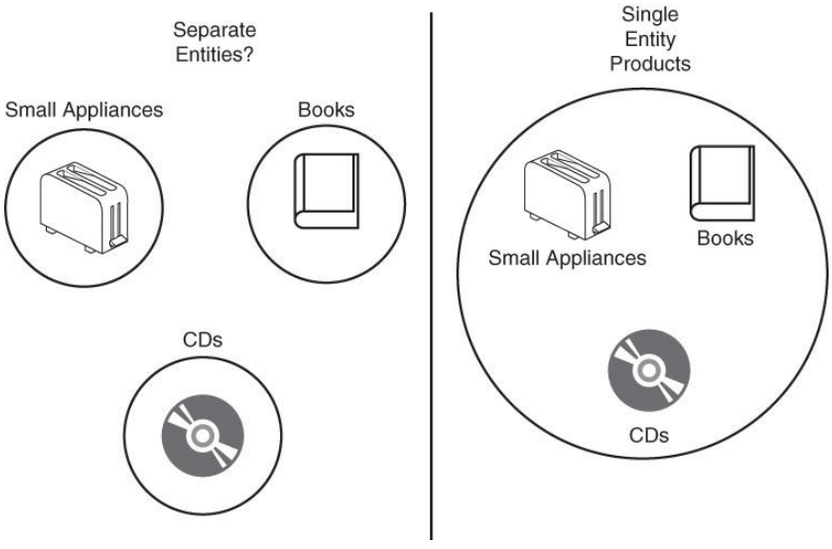
Scenario: in addition to tracking customers and their clickstream data, you would like to track the products customers have ordered. You have decided the first step in this process is to create a document collection containing all products, which for our purposes, includes books, CDs, and small kitchen appliances. There are only three types of products now, but your client is growing and will likely expand into other product types as well.

All of the products have the following information associated with them:	Each of the product types will have specific fields. Books will have fields with information about
• Product name • Short description • SKU (stock keeping unit) • Product dimensions • Shipping weight • Average customer review score • Standard price to customer • Cost of product from supplier	• Author name • Publisher • Year of publication • Page count
The CDs will have the following fields:	The small kitchen appliances will have the following fields:
• Artist name • Producer name • Number of tracks • Total playing time	• Color • Voltage • Style

How should you go about deciding how to organize this data into one or more document collections? Start with how the data will be used. Your client might tell you that she needs to be able to answer the following queries:

- What is the average number of products bought by each customer?
- What is the range of number of products purchased by customers (that is, the lowest number to the highest number of products purchased)?
- What are the top 20 most popular products by customer state?
- What is the average value of sales by customer state (that is, Standard price to customer – Cost of product from supplier)?
- How many of each type of product were sold in the last 30 days?

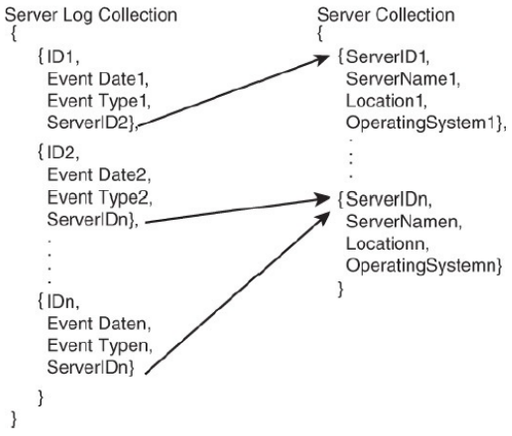
All the queries use data from all product types, and only the last query subtotals the number of products sold by type. This is a good indication that all the products should be in a single document collection. Unlike the example of the collection with clickstream and server log data, the product document types are frequently used together to respond to queries and calculate derived values



Another reason to favor a single document collection is that the client is growing and will likely add new product types. If the number of product types grows into the tens or even hundreds, the number of collections would become unwieldy.

NORMALIZATION OR DENORMALIZATION?

Database normalization is the process of organizing data into tables in such a way as to reduce the potential for data anomalies. An anomaly is an inconsistency in the data. Normalization reduces the amount of redundant data in the database (needed join operation). Rather than repeat customer names and addresses with each order, those attributes would be placed in their own tables. Additional attributes could be associated with both customers and addresses.



Normalized documents imply that you will have references to other documents so you can look up additional information. For example, a server log document might have a field with the identifier of the server that generates log event data. A collection of server documents would have additional information about each server so it does not have to be repeated in each log event document

It can cause performance problems. This is especially true if you have to look up data in two or more large tables. This process is called joining, and it is a basic operation in relational databases. A great deal of effort has gone into developing efficient ways to join data. Database administrators and data modelers can spend substantial amounts of time trying to improve the performance of join operations. It does not always lead to improvement.

You could design a highly normalized database with no redundant data but suffer poor performance. When that happens, many designers turn to denormalization.

Denormalization introduces redundant data. You might wonder, why introduce redundant data? It can cause data anomalies. It obviously requires more storage to keep redundant copies of data. The reason to risk data anomalies and use additional storage is that denormalization can significantly improve performance.

When data is denormalized, there is no need to read data from multiple tables and perform joins on the data from the multiple collections. Instead, data is retrieved from a single collection or document. This can be much faster (improve read operation) than retrieving from multiple collections, especially when indexes are available.

COLUMN DATABASE

- OLTP - Online Transaction Processing: software architectures oriented towards handling ACID transactions.
    - is transactional system and deals with the operation in a system with lot of short transactions on-line i.e. INSERT, UPDATE, DELETE
    - OLTP focus on very fast query processing
    - quite efficient to maintain data in multi-accessed environments
    - data is frequently updated
- OLAP - Online Analytics Processing: software architectures oriented towards interactive and fast analysis of data. Typical of Business Intelligence Software.

- is analytical system and deals with historical data with low volume of transactions
- Response time is an effective measure of the OLAP systems
- Queries are quite complex
- processing speed depends upon the amount of data involved

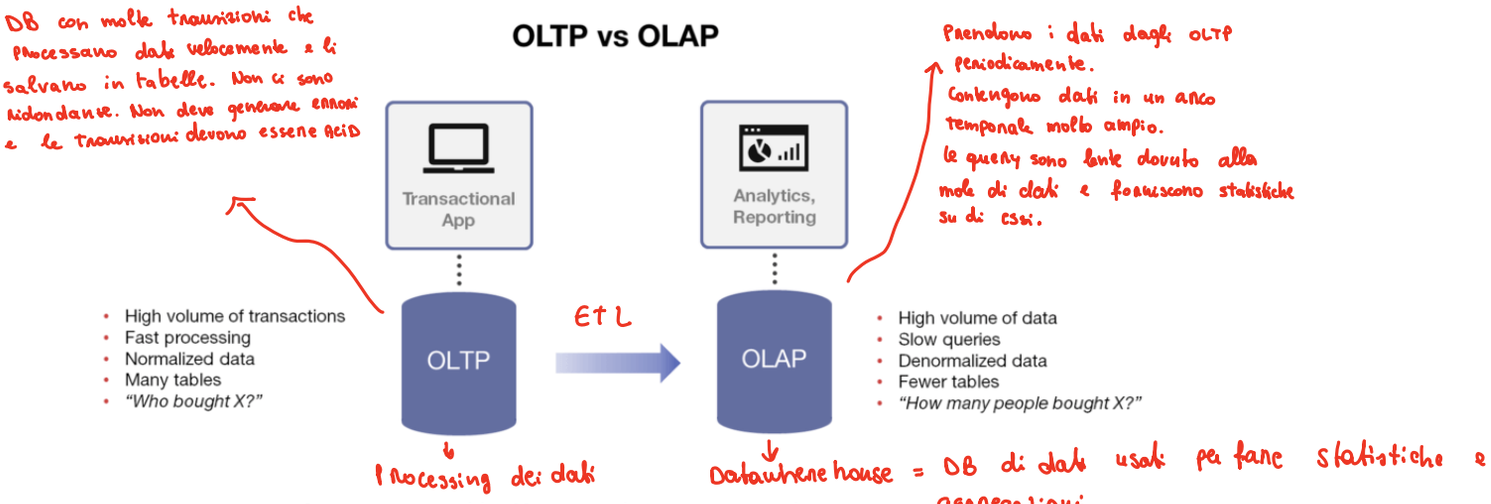


Image extracted from: https://diffzi.com/oltp-vs-olap/

ROW DATA ORGANIZATION

Since the beginning of digital files, the data of each record were physically organized in rows.

- OLTP processing is mainly oriented towards handling one record at a time processing.

When the first relational databases were designed, the world was mainly experiencing the OLTP era.

The record-oriented workflow handled by the first relational databases and the row-oriented physical structure of early digital files provided good performance

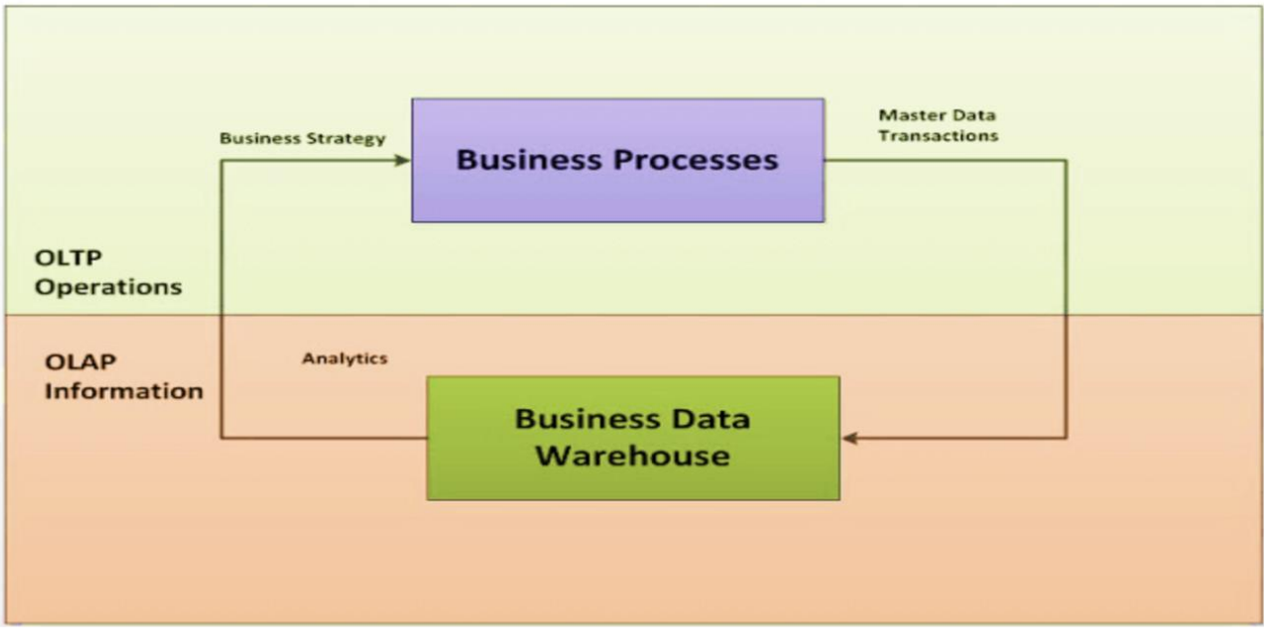
CRUD AND QUERIES

In the record-based processing era (up to the end of the 80s of the last century), CRUD operations (Create, Read, Update, Delete) were the most time-critical ones.

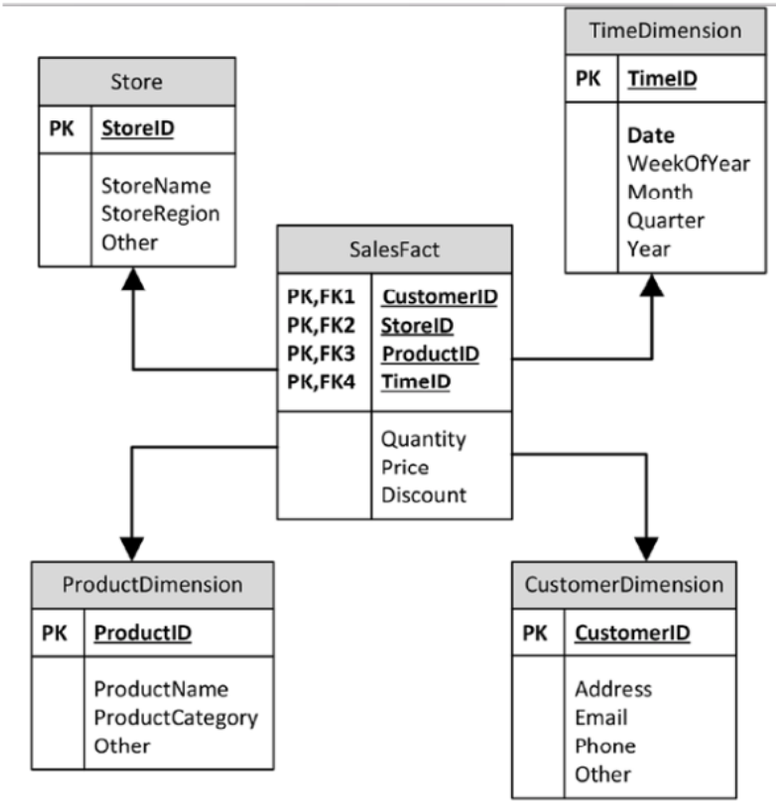
Reporting programs were typically iterated through entire tables and were run in a background batch mode. The time response to the queries was not a critical issue.

When Business Intelligence software started to spread, OLAP processing assumed a big relevance, thus the time response to queries became a critical issue to appropriately handle

Data warehouses: relational databases tasked with supporting analytic and decision support applications (80's-90's era).



STAR SCHEMAS IN DATA WAREHOUSE



Star schemas were developed to create data warehouses in which **aggregate queries** could **execute quickly** and which would provide a predictable schema for Business Intelligence (BI) tools. In a star schema, **central large fact tables** are associated with numerous smaller dimension tables. When the dimension tables implement a more complex set of foreign key relationships, then the schema is referred to as a snowflake schema.

Although star schemas are typically found in relational databases, they do not represent a fully normalized relational model of data. Both the dimension tables and the fact tables usually contain redundant information, or information depended only partly on the primary key. In some respects, widespread adoption of the star schema was a step away from full compliance with Codd's relational model.

MAIN DRAWBACKS

Almost all data warehouses adopted some variation on the star schema paradigm, and almost all relational databases adopted indexing and SQL optimization schemes to accelerate queries against star schemas. These optimizations allowed relational data warehouses to serve as the foundation for management dashboards and popular BI products. However, despite these optimizations:

- Data processing in data warehouses remained severely CPU and IO intensive.
- As **data volumes** grew and user demands for interactive response times continued
- there was **increasing** discontent with traditional data warehousing performance.

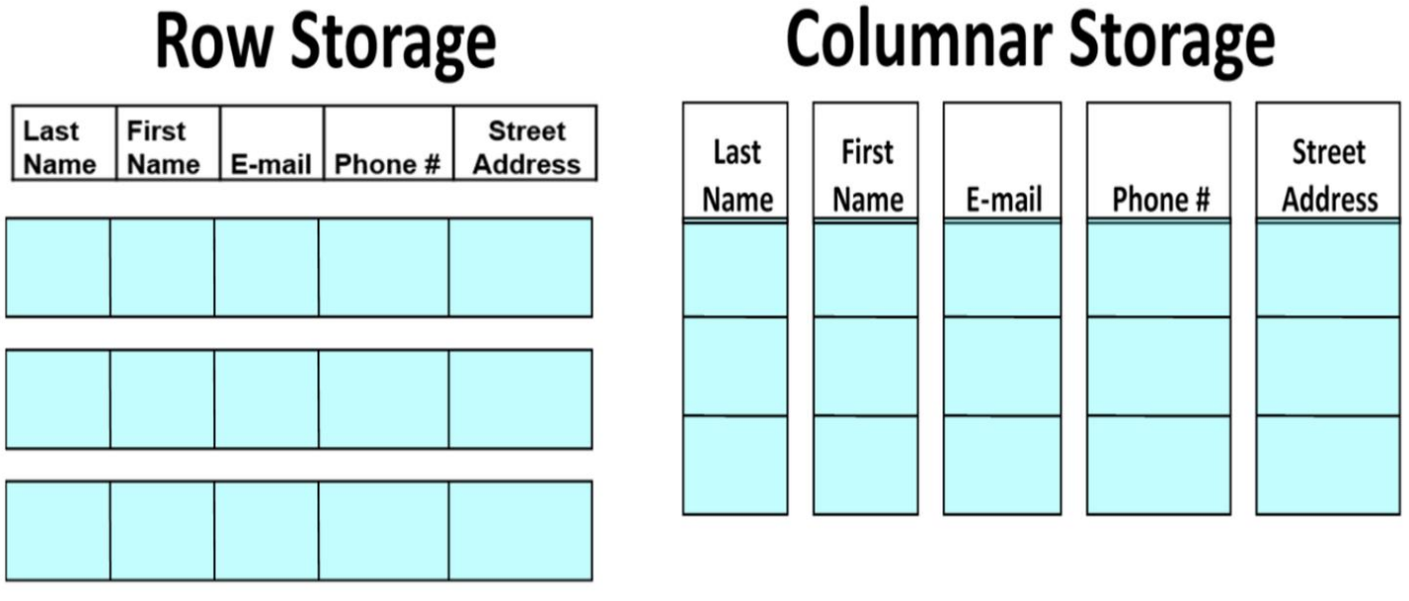
THE COLUMNAR STORAGE

When processing data for making **analytics**, in most of cases, we are not interested in retrieving all the information of each single records.

Indeed, we are interested, for example, in retrieving the values of one attribute of a set of records (for making trend graphs, or calculating statistics).

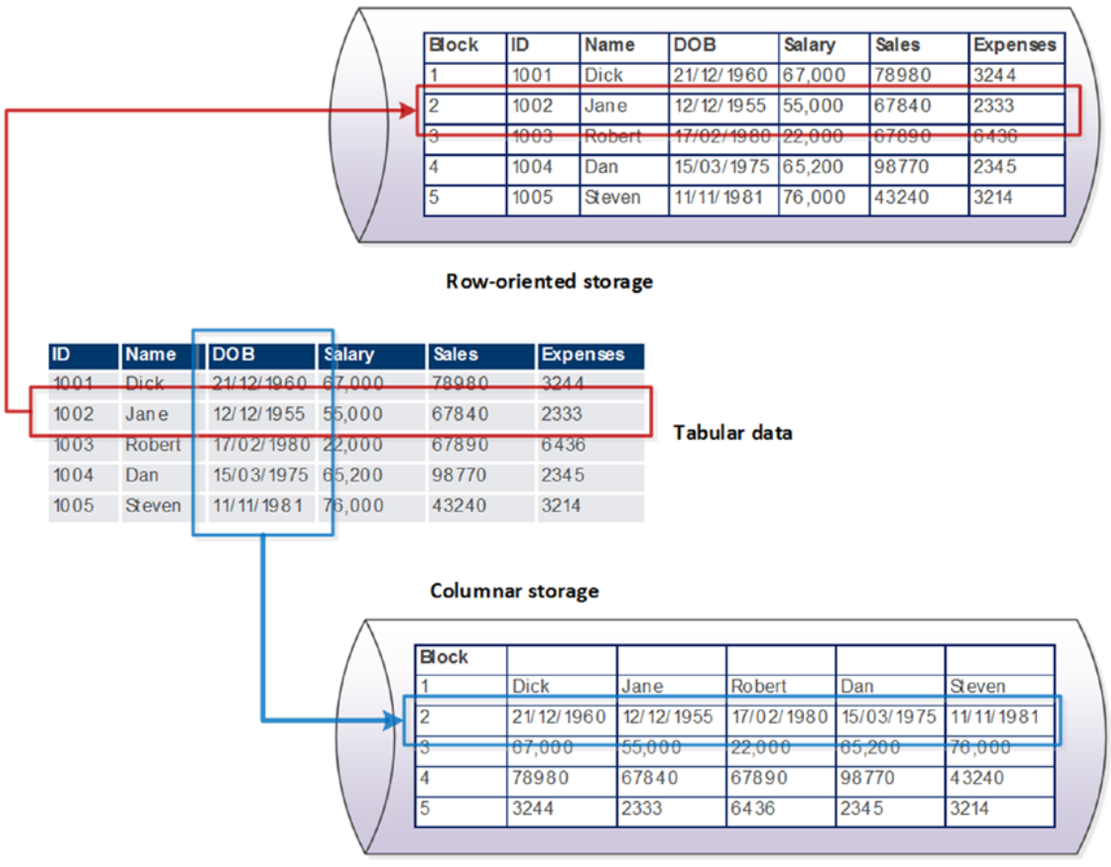
When dealing with row-based storage of records, we have to access to all the records of the considered set for retrieving just the values of one attribute.

If all the values of an attribute are grouped together on the disk (or in the block of a disk), the task of retrieving the values of one attribute will be faster than a row-based storage of records.

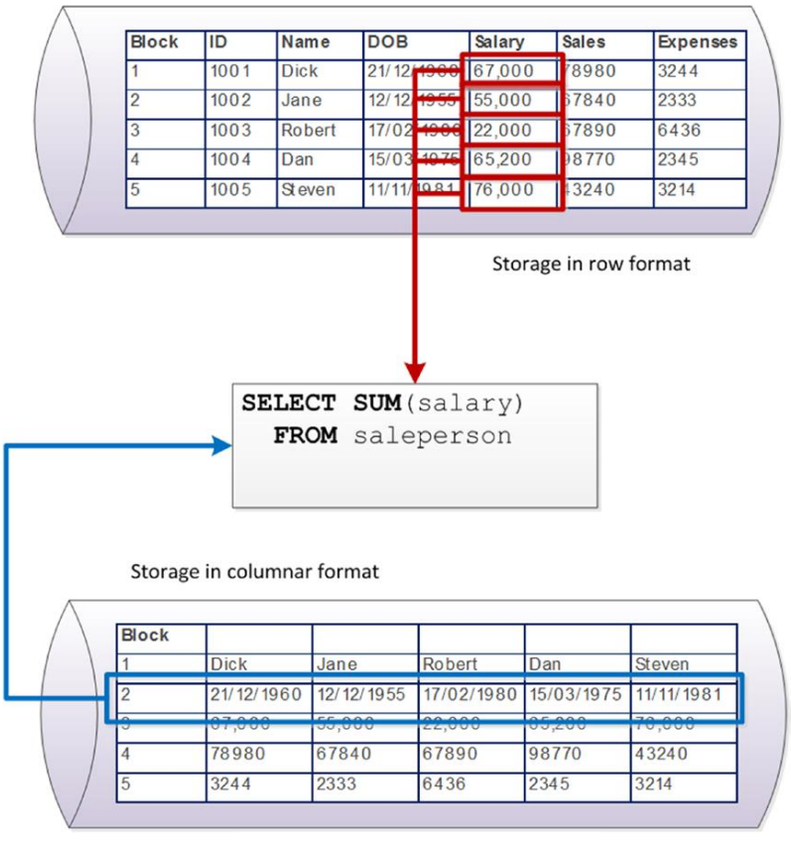


The idea that it might be better to store data in columnar format dates back to the 1970s, although commercial columnar databases did not appear until the mid-1990s.

The essence of the columnar concept is that data for columns is grouped together on disk. In a columnar database, values for a specific column become co-located in the same disk blocks, while in the row-oriented model, all columns for each row are co-located.



There are two big advantages to the columnar architecture. First, in a columnar architecture, queries that seek to aggregate the values of specific columns are optimized, because all of the values to be aggregated exist within the same disk blocks. This phenomenon for our sample database; retrieving the sum of salaries from the row store must scan five blocks, while a single block access suffices in the column store.



The exact IO and CPU optimizations delivered by a column architecture vary depending on workload, indexing, and schema design. In general, queries that work across multiple rows are significantly accelerated in a columnar database.

COLUMNAR COMPRESSION

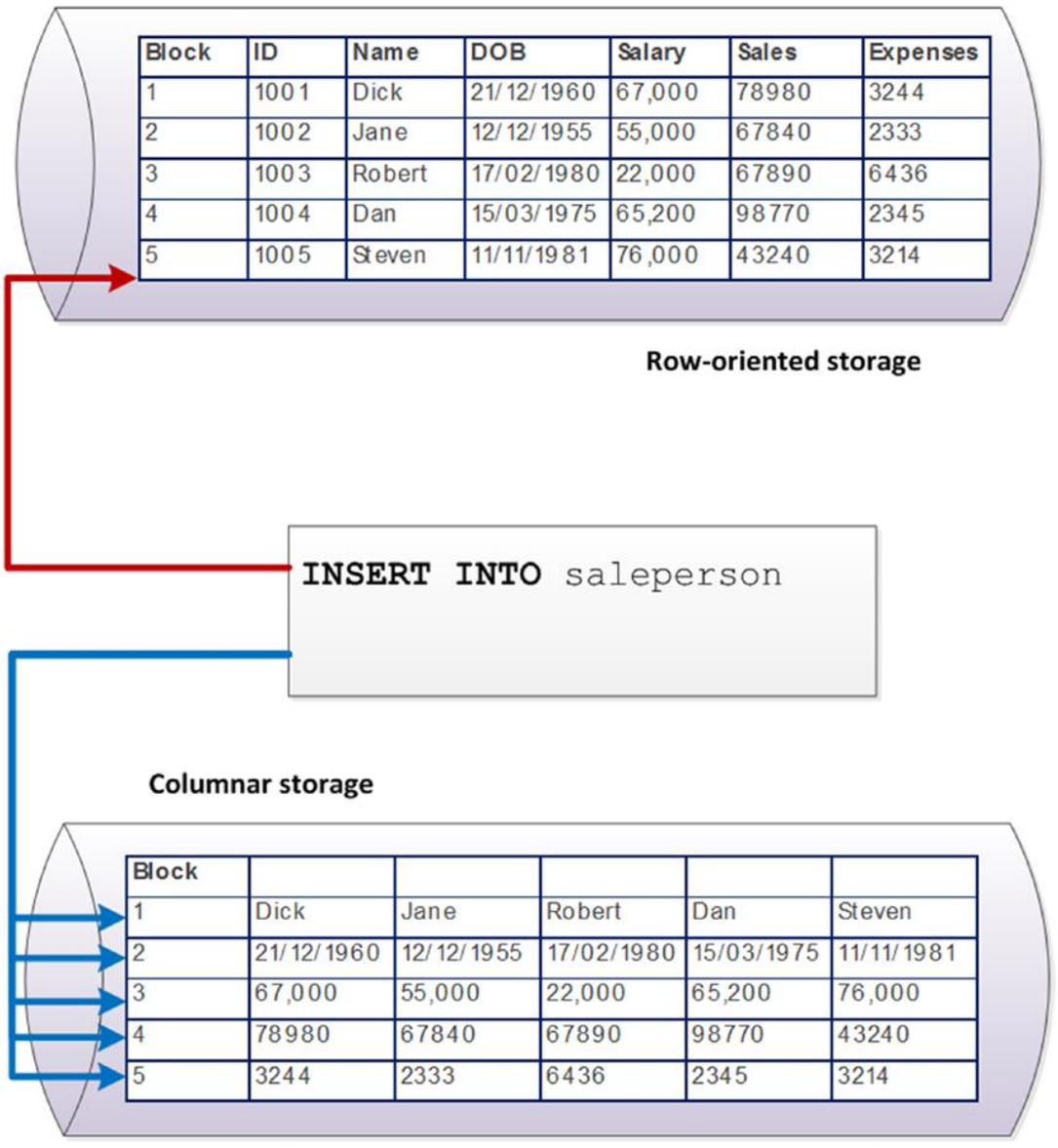
The second key advantage for the columnar architecture is compression. Compression algorithms work primarily by removing redundancy within data values. Data that is highly repetitious—especially if those repetitions are localized—achieve higher compression ratios than data with low repetition. Although the total amount of repetition is the same across the entire database, regardless of row or column orientation, compression schemes usually try to work on localized subsets of the data; the CPU overhead of compression is far lower if it can work on isolated blocks of data. Since in a columnar database the columns are stored together on disk, achieving a higher compression ratio is far less computationally expensive.

Furthermore, in many cases a columnar database will store column data in a sorted order. In this case, very high compression ratios can be achieved simply by representing each column value as a "delta" from the preceding column value. The result is extremely high compression ratios achieved with very low computational overhead. -> Perché i valori delle colonne vengono salvati insieme nel disco quindi è più facile accedere a tutti i valori e fare la compressione

COLUMNAR WRITE PENALTY

The key disadvantage of the columnar architecture—and the reason it is a poor choice for OLTP databases – is the overhead it imposes on single row operations. In a columnar database, retrieving a single row involves assembling the row from each of the column stores for that table. Read overhead can be partly reduced by caching and multicolumn projections (storing multiple columns together on disk). However, when it comes to DML operations—particularly inserts—there is virtually no way to avoid having to modify all the columns for each row.

The insert overhead for a column store on our simple example database. The row store need only perform a single IO to insert a new value, while the column store must update as many disk blocks as there are columns.



In real-world row stores (e.g., traditional RDBMS), inserts require more than a single IO because of indexing overhead, and real-world column stores implement mitigating schemes to avoid the severe overhead during single-row modifications. However, the fundamental principle—that column stores perform poorly during single-row modifications—is valid.

DELTA STORE

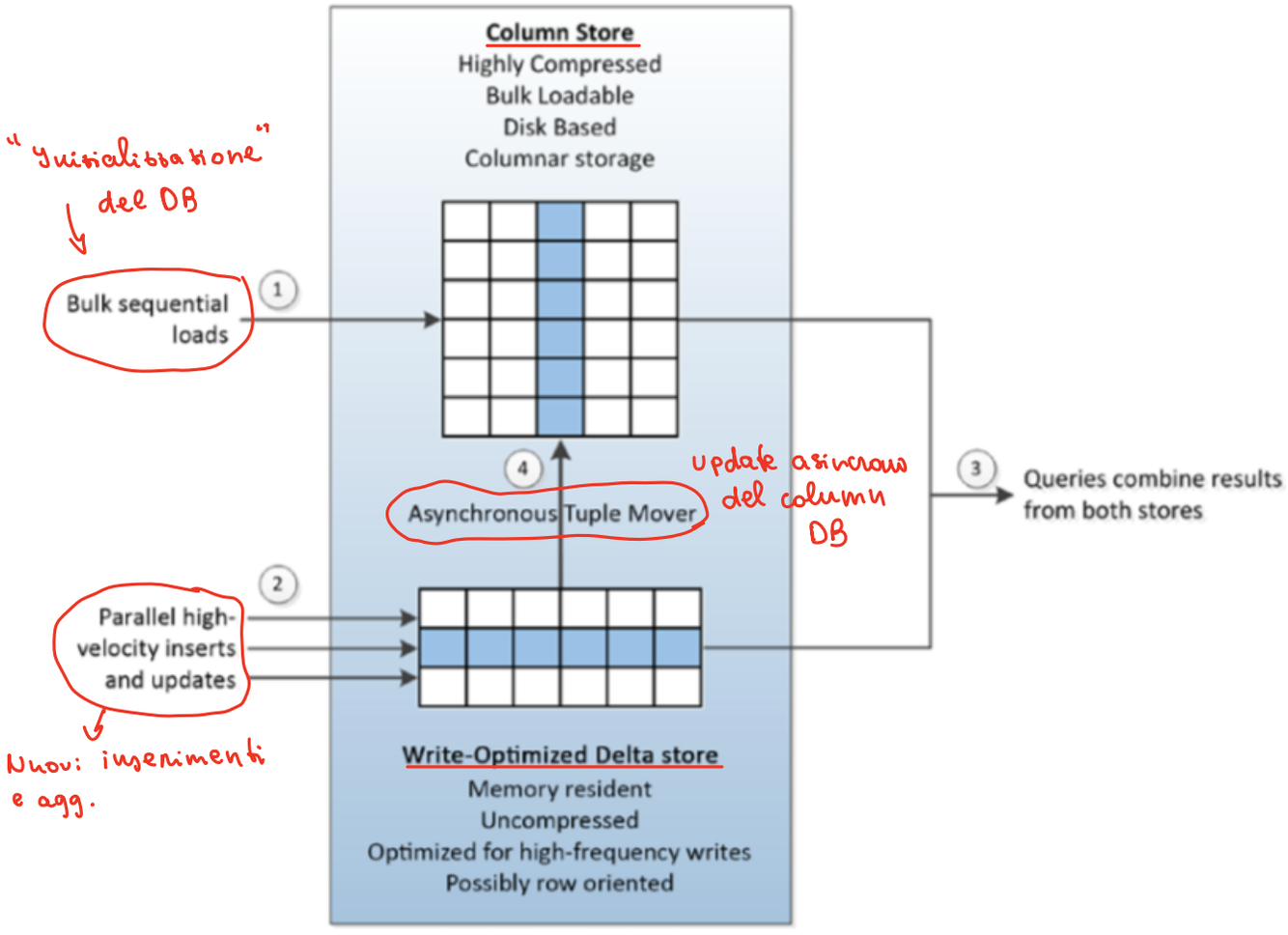
As we noted earlier, insert and update overhead for single rows is a key weakness of a columnar architecture. Many data warehouses were bulk-loaded in daily batch jobs—a classic overnight ETL scenario. However, it became increasingly important for data warehouses to provide real-time "**up-to-the-minute**" information, which implied that the data warehouse should be able to accept a constant trickle feed of changes.

PROBLEMS: The simplistic columnar architecture outlined above would be **unable to cope with this constant stream of row-level modifications**.

To address this issue, columnar databases generally implement **some form of write-optimized delta store** (we'll call this the delta store for short). This area of the database is optimized for frequent writes. You can think simplistically of the data in the delta store as being in a row format, although in practice the internal format might still be columnar or a row/column hybrid.

SOLUTION: Regardless of the internal format of the data, the **delta store is generally memory resident**, the data is **generally uncompressed**, and the store can accept high-frequency data modifications.

Data in the delta store is **periodically merged** with the main columnar-oriented store. In Vertica, this process is referred to as the Tuple Mover and in Sybase IQ as the RLV (Row Level Versioned) Store Merge. The merge will occur periodically, or whenever the amount of data in the delta store exceeds a threshold. Prior to the merge, queries might have needed to access both the delta store and the column store in order to return complete and accurate results.



The figure shows a generic **columnar database architecture**. The database contains a primary column store that contains highly compressed columnar data backed by disk storage. A smaller write-optimized delta store contains data that is minimally compressed, memory resident, and possibly row oriented.

1) Large-scale bulk sequential loads (Nightly Extract, Transform, Load – **ETL** jobs) will generally be directed to the column store
2) **Incremental** inserts and updates will be directed to the write-optimized store
3) Queries may need to read from both stores in order to get **complete** and **consistent** results
4) Periodically, or as required, a process will **shift data** from the write-optimized store to the column store.

PROJECTIONS

In the simplistic description of a columnar database earlier, we showed each column stored together. For queries that access only a single column, storing each column in its own region on disk may be sufficient, but

in practice complex queries need to read combinations of column data. It therefore sometimes **makes sense to store combinations of columns together on disk**.

To achieve this, columnar databases such as Vertica store tables physically as a series of **projections**, that is combinations of columns that are frequently accessed together.

For instance, in figure we see a single logical table with three projections. In Vertica,

1) each table has a default superprojection that includes all the columns in the table
2) Additional projections are created to support specific queries
    a. In this case, a projection is created to support sales aggregated by customer
3) another projection created for sales aggregated by region and product

Region	Customer	Product	Sales
A	G	C	789
B	C	C	743
D	F	D	675
C	C	A	23
A	R	B	654

Logical Table
Table appears to user in relational normal form

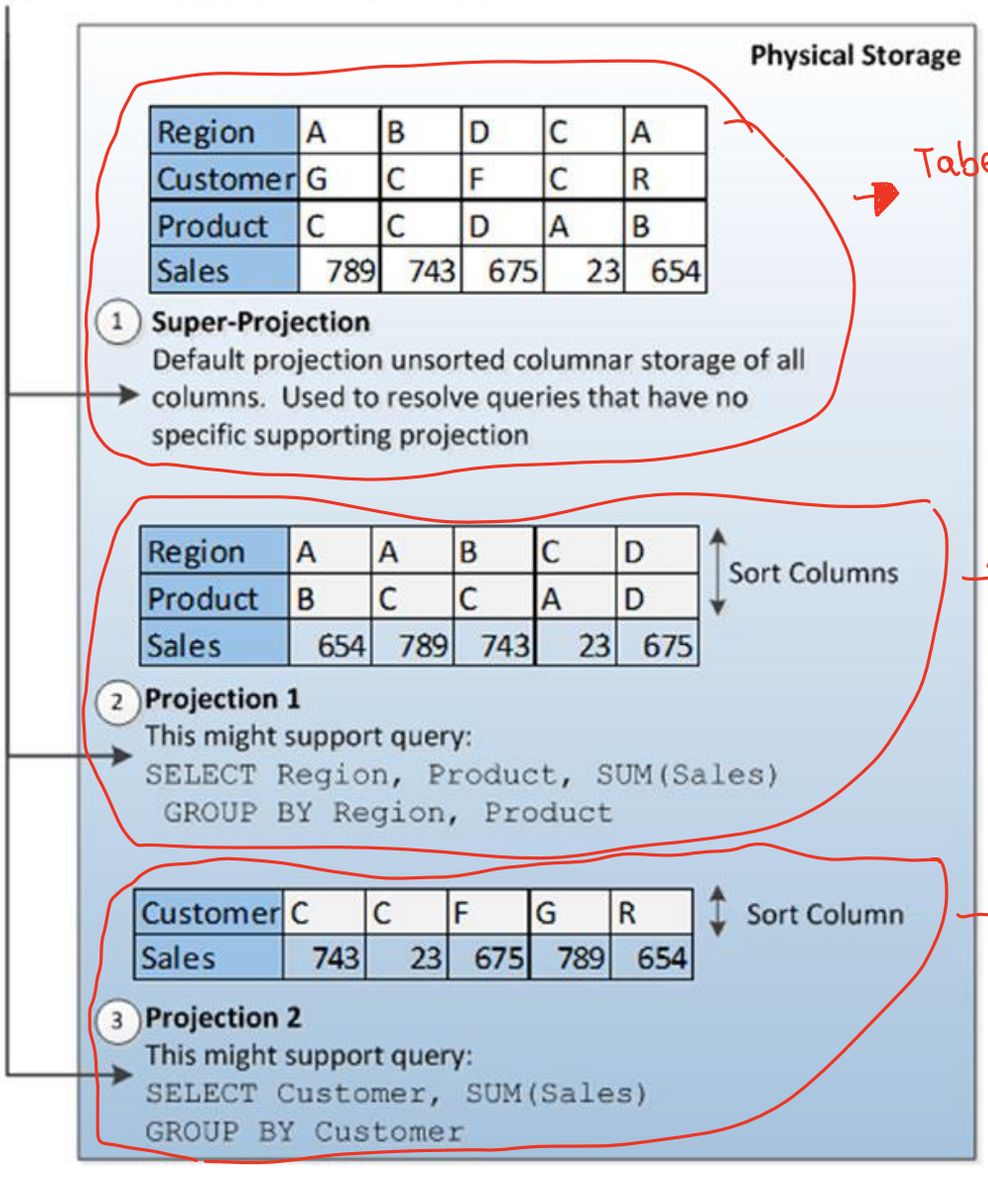


Tabella "normale" (intera)

Proiezione sulle colonne customer, product, sales

Proiezione sulle colonne customer, sales

In Vertica, projections may be sorted on one or more columns. This decreases processing time for sort and aggregate operations, and also increases compression efficiency. Vertica also supports pre-join projections that materialize columns from multiple tables based on a join criterion. Pre-join projections serve a similar function to materialized views that are created to support specific join operations in traditional relational systems.

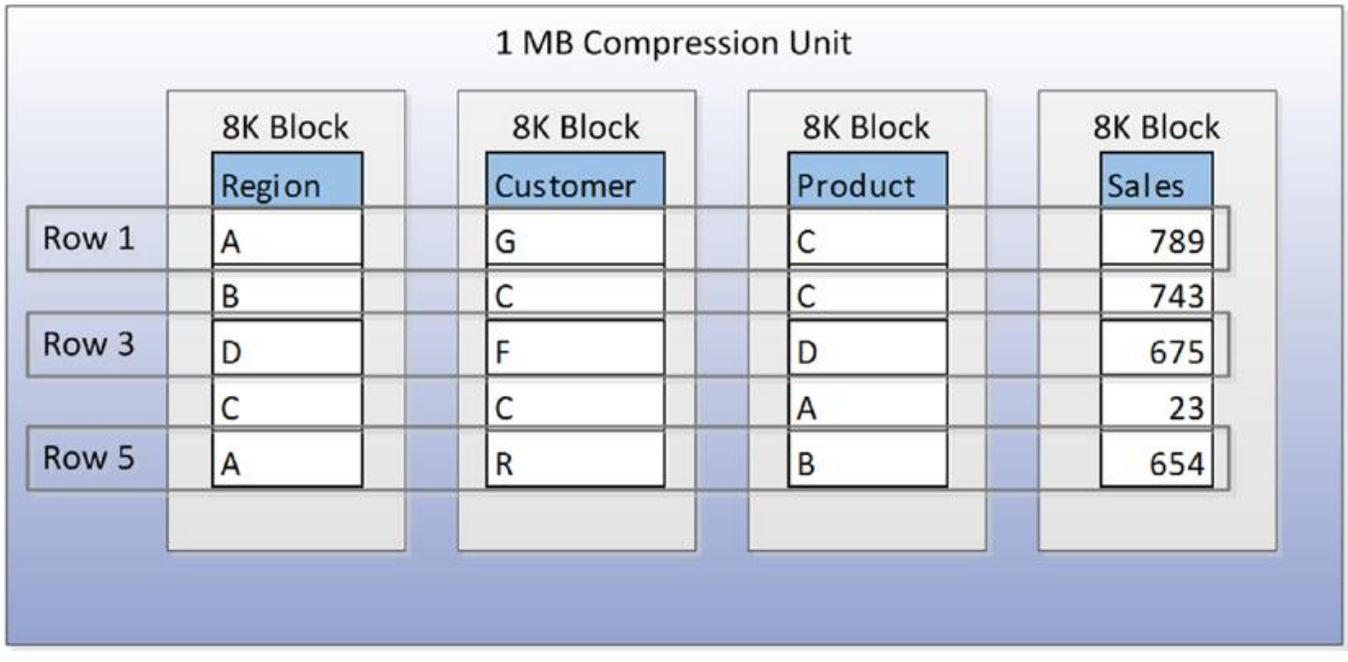
Projections may be created manually by the database administrator, on the fly by the query optimizer in response to a specific SQL, or in bulk based on historical workloads. Creating the correct set of projections is of critical importance to columnar store performance—roughly equivalent to the importance of correct indexing in a row store.

Sybase IQ refers to its query optimization structures as "indexes"; however, these indexes bear more resemblance to Vertica projections than to B-Tree indexes in Oracle or SQL Server. Indeed, the Sybase IQ default index—which is automatically created on all columns during table creation—is called the fast projection index. Sybase also supports other indexing options based on traditional bitmap and B-Tree indexing schemes.

HYBRID COLUMNAR COMPRESSION – ORACLE SOLUTION

Variations on the columnar paradigm have been implemented within **both traditional relational systems** and other "NewSQL" systems. For instance the in-memory database SAP HANA provides support for column or row orientation on a table-by-table basis. The Oracle 12c "Database in Memory" also implements a column store.

Oracle's Enhanced Hybrid Columnar Compression (EHCC) is an interesting attempt to achieve a best-of-both-worlds combination of row and column storage technologies. In EHCC— currently only available in Oracle's Exadata system—rows of data are contained within compression units of about 1 MB, with columns stored together within smaller 8K blocks. Because the columns are stored together within blocks, high levels of compression can be achieved. Because rows are guaranteed to be within a 1 MB compression unit, the overhead for performing row-level modifications is reduced. Figure 6-7 illustrates the concept. Rows are contained within 1MB compression units, but each 8K block contains data for a specific column that is highly compressed.



Column-oriented storage is common in modern nonrelational systems as well. Apache Parquet is a column-oriented storage mechanism for Hadoop files that allows Hadoop systems to take advantage of columnar performance advantages and compression. Apache Kudu uses both row and column storage formats to bridge the perceived performance gap between HDFS row-based processing and Hadoop file scans.

KEY-VALUE AND DOCUMENT DATABASES: SOME ISSUES

NoSQL databases discussed so far, may help us to solve the problem of handling huge amount of data.

However, some issues occur:

- Key-value databases lack support for organizing many columns and keeping frequently used data together.
- Document databases may not have some of the useful features of relational databases, such as a SQL-like query language.

BIGTABLE: THE GOOGLE NOSQL DATABASE

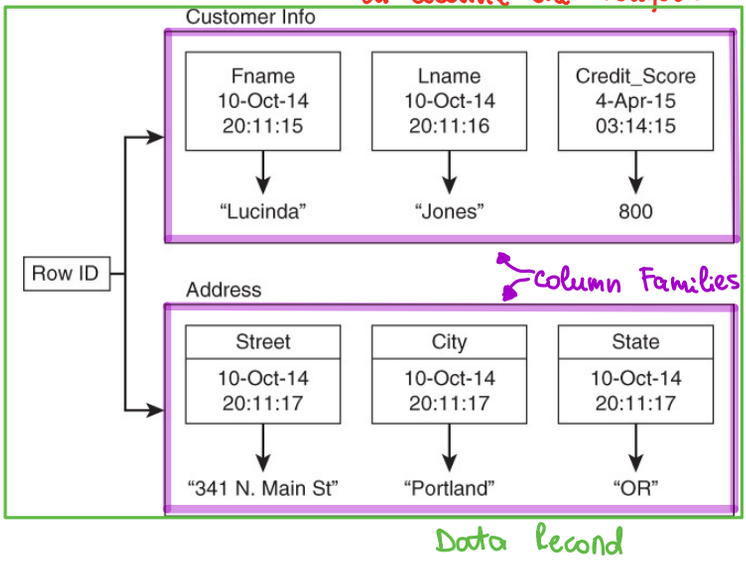
It was introduced in 2006 for Google large services such as Google Earth and Google Finance, and it belongs to column database family.

Main Features:

- Column based with a dynamic control over columns
- Data values are indexed by row identifier, column name, and a time stamp (quest'ultimo è importante per mantenere le informaione anche passate per ogni record)
- Reads and writes of a row are atomic (se una modifica avviene su più attribute sono sicuro vhr in caso di fallimento si mantenga la consistenza)
- Rows are maintained in a sorted order

EXAMPLE

-> Gli attr di una riga vengono suddivisi in column family, le quali contengono un insieme di colonne che vengono usate solitamente insieme.



A data record is a row composed of several column families.

Each family consists of a set of related columns (frequently used together.)

Column values within a column family are kept together on the disk, whereas values of the same row belonging to different families may be stored far.

Columns families must be defined a priori (like relational tables).

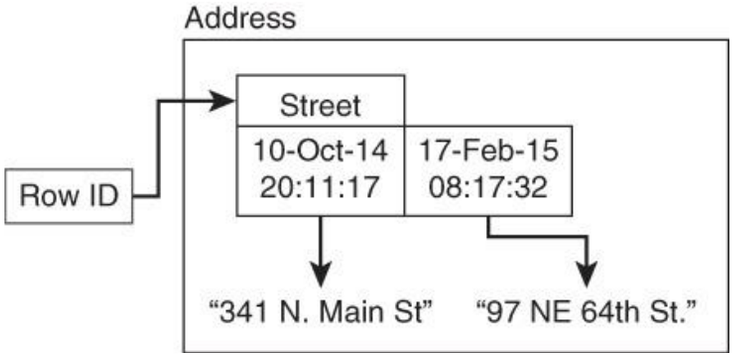
Applications may add and remove columns inside families (like key-value pairs).

INDEXING

In BigTable, a data value is indexed by

- **row identifier** → similar role than a primary key in relational databases
- **column name** → identifies a column (similar role than a key in key-value databases)
- **time stamp** → which allows the existence of multi versions of a column value can exist
    - when a new value is written to a BigTable database, the old value is not overwritten. Instead, a new value is added along with a time stamp

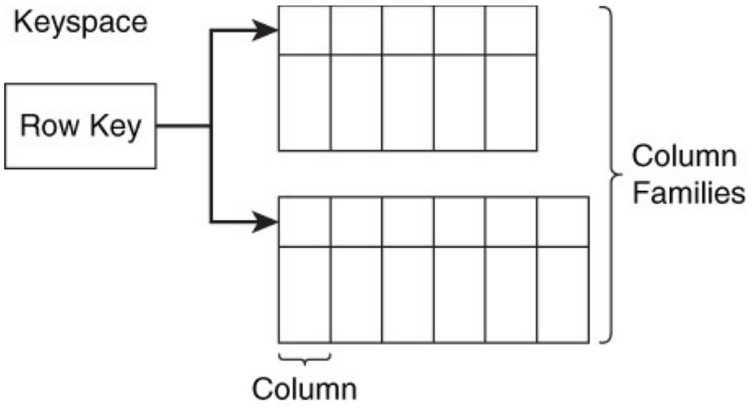
It uniquely identifies a row. Remember, a single row can have multiple column families. Unlike row-oriented relational databases that store all of a row's data values together, column family databases store only portions of rows together.



BASIC COMPONENTS OF COLUMN FAMILY DATABASES

KEYSPACE

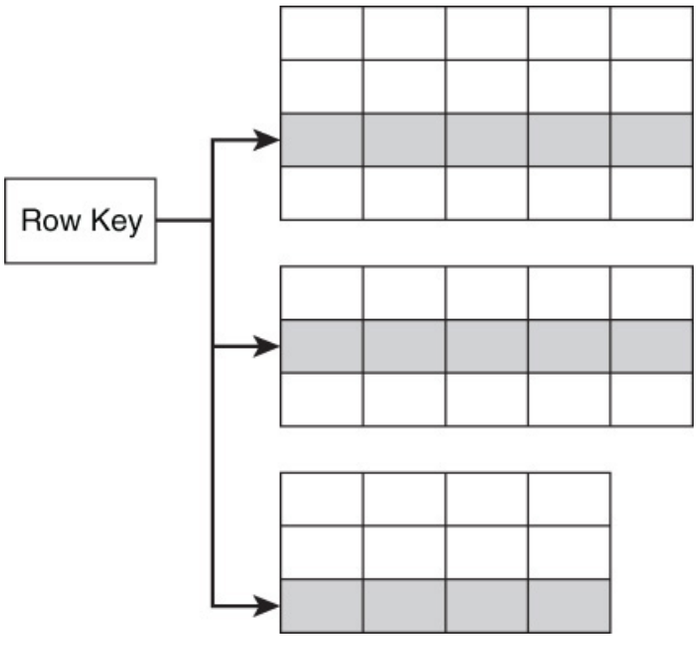
A keyspace is the top-level data structure (**a sort of container**) in a column family database. It is top level in the sense that all other data structures you would create as a database designer are contained within a keyspace. A keyspace is analogous to a schema in a relational database. Typically, you will have one keyspace for each of your applications.



It holds column families, row keys, and related data structures. Typically, there is one keyspace per application.

ROW KEY

Row keys are one of the components used to uniquely identify values stored in a column family database. It serves some of the same purposes as a primary key in a relational database.



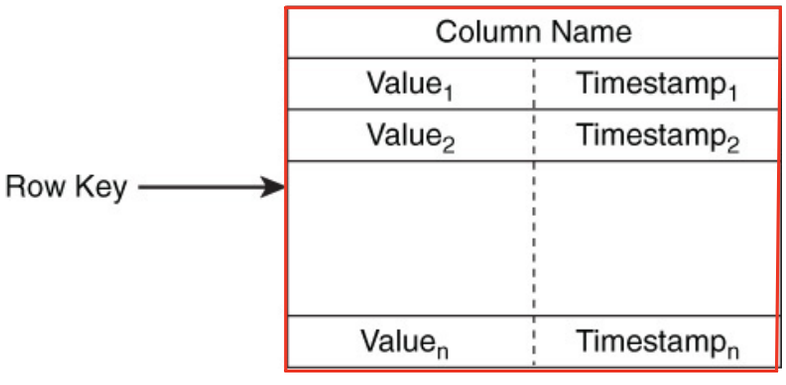
The others are column family names, column names, and a version ordering mechanism (time stamp).

In addition to uniquely identifying rows, row keys are also used to partition and order data. In HBase, rows are stored in lexicographic order of row keys. You can think of this as alphabetic ordering with additional orderings for nonalphabetic characters.

In Cassandra, rows are stored in an order determined by an object known as a partitioner. Cassandra uses a random partitioner by default. As the name implies, the partitioner randomly distributes rows across nodes. Cassandra also provides an order-preserving partitioner, which can provide lexicographic ordering.

COLUMNS

A column is the **data structure for storing a single value** in a database. Depending on the type of column family database you are using, you might find values are represented simply as a string of bytes. This minimizes the overhead on the database because it does not validate data types, like in BigTable.



Valori nel tempo di quella specifica row-key sulla specifica colonna Column-Name. Vengono salvati tutti i valori, non vengono sostituiti man mano nel tempo (sovrascritti)

In other cases, you might be able to specify data types ranging from integers and strings to lists and maps. Cassandra's Query Language (CQL) offers almost 20 different data types. Values can vary in length. For example, a value could be as simple as a single integer, such as 12, or as complex as a highly structured XML document.

Columns are members of column family databases. Database designers define column families when they create a database. However, developers can add columns any time after that. Just as you can insert data into a relational table, you can create new columns in column family databases.

Columns have three parts, uniquely identifies values:

1. A column name
2. A time stamp or other version stamp

3. A value

The column name serves the same function as a key in a key-value pair: It refers to a value. The time stamp or other version stamp is a way to order values of a column. As the value of a column is updated, the new value is inserted into the database and a time stamp or other version stamp is recorded along with the column name and value. The version mechanism allows the database to store multiple values associated with a column while maintaining the ability to readily identify the latest value. Column family databases vary in the types of version control mechanisms used.

COLUMN FAMILIES

Column families are **collections of related columns**. Columns that are frequently used together should be grouped into the same column family. For example, a customer's address information(street, city, state, and zip code) should be grouped together in a single column family.

Column families are stored in a keyspace. Each row in a column family is uniquely identified by a row key. This makes a column family analogous to a table in a relational database. There are important differences, however. Data in relational database tables is not necessarily maintained in a predefined order. Rows in relational tables are not versioned the way they are in column family databases.

Street	City	State	Province	Zip	Postal Code	Country
178 Main St.	Boise	ID		83701		U.S.
89 Woodridge	Baltimore	MD		21218		U.S.
293 Archer St.	Ottawa		ON		K1A 2C5	Canada
8713 Alberta DR	Vancouver		BC		VSK 0AI	Canada

Column families are analogous to relational database tables: They store multiple rows and multiple columns. There are, however, significant differences between the two, including varying columns by row.

Perhaps most importantly, columns in a relational database table are not as dynamic as in column family databases. Adding a column in a relational database requires changing its schema definition. Adding a column in a column family database just requires making a reference to it from a client application, for example, inserting a value to a column name.

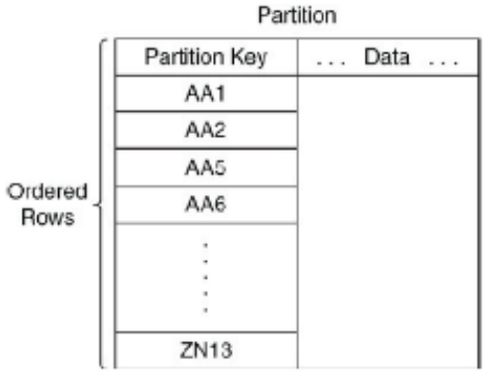
CLUSTER AND PARTITIONS

The most recent column databases are designed for ensuring high availability and **scalability**, along with different **consistency levels** and data storage across a set of servers.

PARTITION

A partition is a **logical subset** of a database. Partitions are usually used to store a set of data based on some attribute of the data in column databases, they can be generated on the basis of:

- A range of values, such as the value of a row ID
- A hash value, such as the hash value of a column name
- A list of values, such as the names of states or provinces
- A composite of two or more of the above options



Each node or server within a column family cluster maintains one or more partitions.

When a client application requests data, the request is routed to a server with the partition containing the requested data. A request could go to a central server in a master-slave architecture or to any server in a peer-to-peer architecture. In either case, the request is forwarded to the appropriate server.

In practice, multiple servers may store copies of the same partition. This improves the chances of successfully reading and writing data even in the event of server failures. It can also help improve performance because all servers with copies of a partition can respond to requests for data from that partition. This model effectively implements load balancing.

CLUSTER

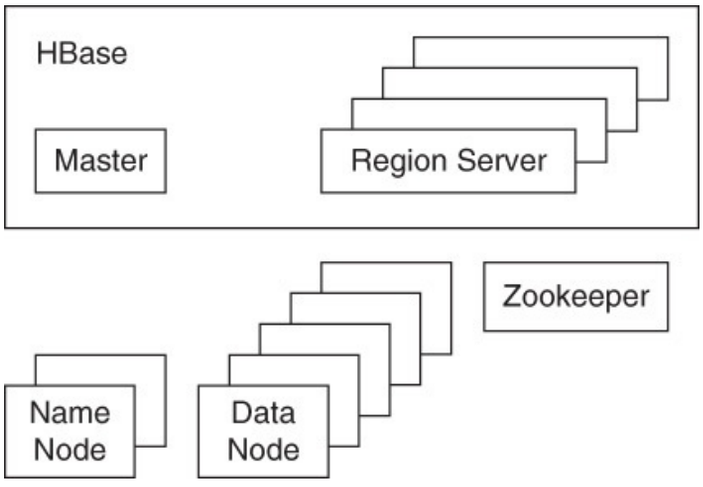
A cluster is a set of servers configured to function together and to implement a distributed service. Servers sometimes have differentiated functions and sometimes they do not.

ARCHITECTURES IN COLUMN FAMILY DATABASE

There are two commonly used types of architectures used with distributed databases: multiple node type (at least two types of nodes, although there may be more) and peer-to-peer type.

1) VARIETY OF NODES – HBASE ARCHITECTURE

HBase is a part of the Hadoop infrastructure. It uses the various types of servers to implement the functional requirements of Hadoop (out of our interest).



The Hadoop File System (HDFS) uses a **master-slave architecture** that consists of name nodes and data nodes. The name nodes manage the file system and provide for centralized metadata management. Data nodes actually store data and replicate data as required by configuration parameters.

Zookeeper is a type of node that enables coordination between nodes within a Hadoop cluster. Zookeeper maintains a shared hierarchical namespace. Because clients need to communicate with Zookeeper, it is a potential single point of failure for HBase. Zookeeper designers mitigate risks of failure by replicating Zookeeper data to multiple nodes.

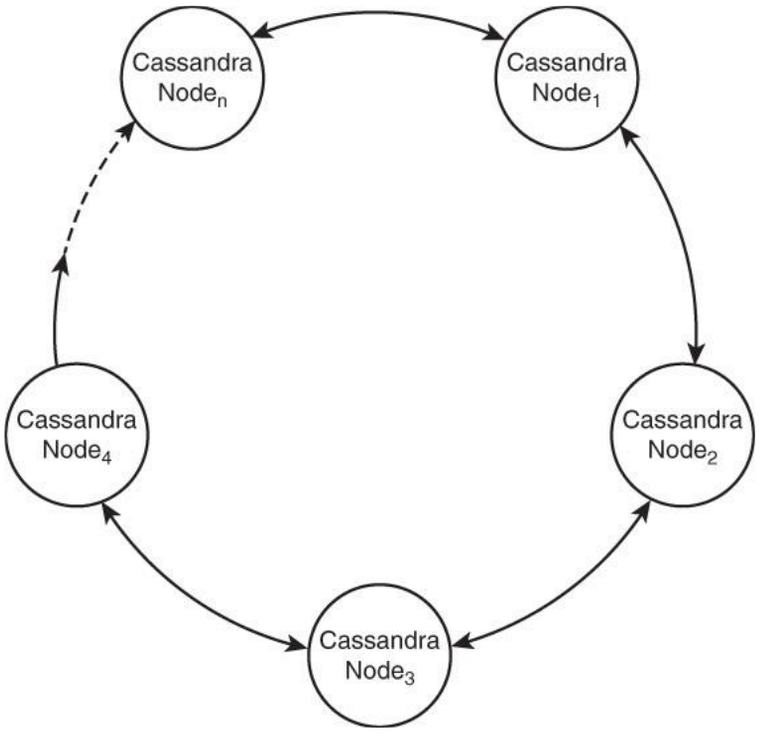
In addition to the Hadoop services used by HBase, the database also has server processes for managing metadata about the distribution of table data. RegionServers are instances that manage Regions, which are storage units for HBase table data. When a table is first created in HBase, all data is stored in a single Region. As the volume of data grows, additional Regions are created and data is partitioned between the multiple Regions. RegionServers, which host Regions, are designed to run with 20–200 Regions per server; each Region should store between 5GB and 20GB of table data. A Master Server oversees the operation of RegionServers.

When a client device needs to read or write data from HBase, it can contact the Zookeeper server to find the name of the server that stores information about the corresponding Region's storage location within the cluster. The client device can then cache that information so it does not need to query Zookeeper again for those device details. The client then queries the server with the Region information to find out which server has data for a given row key (in the case of a read) or which server should receive data associated with a row key (in the case of the write).

An advantage of this type of architecture is that servers can be deployed and tuned for specific tasks, for example, managing the Zookeeper. It does, however, require system administrators to manage multiple configurations and to tune each configuration separately. An alternative approach is to use a single type of node that can assume any role required in the cluster. Cassandra uses this approach.

2) PEER-TO-PEER: CASSANDRA ARCHITECTURE

Apache Cassandra, like Apache HBase, is designed for high availability, scalability, and consistency. Cassandra takes a different architectural approach than HBase. Rather than use a hierarchical structure with fixed functions per server, Cassandra uses a **peer-to-peer model**. All Cassandra nodes run the same software. They may, however, serve different functions for the cluster.



There are several advantages to the peer-to-peer approach

- Simplicity: no node can be a single point of failure.
- Scaling up and down: servers are added or removed from the cluster in a easily way.
  - Servers in a peer-to-peer network communicate with each other and, eventually, new nodes are assigned a set of data to manage.
  - When a node is removed, servers hosting replicas of data from the removed node respond to read and write requests.

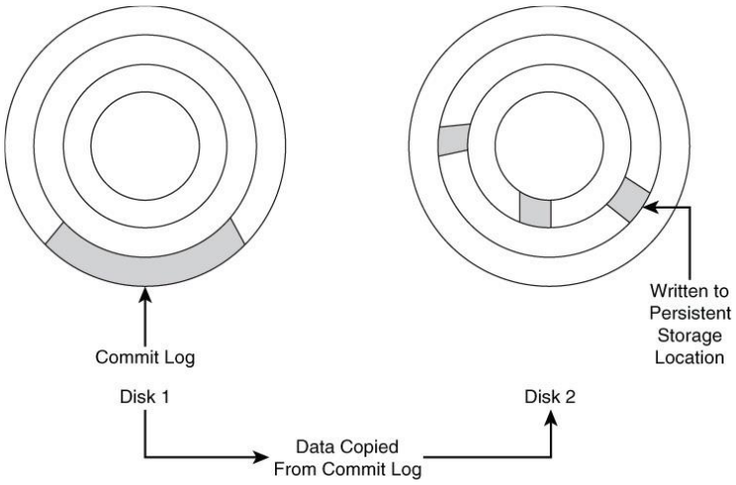
Because peer-to-peer networks do not have a single master coordinating server, the servers in the cluster are responsible for managing a number of operations that a master server would handle so each node shares similar responsibilities, including the following:

- Accepting read and write requests
- Forwarding read and write requests to servers able to fulfill the requests
- Gathering and sharing information about the state of servers in the clusters
- Helping compensate for failed nodes by storing write requests for the failed node until it is restored

Cassandra has protocols to implement all of these functions (Gossip protocol).

COMMIT LOGS

If your application writes data to a database and receives a successful status response, you reasonably expect the data to be stored on persistent storage. Even if a server fails immediately after sending a write success response, you should be able to retrieve your data once the server restarts.



Commit Logs are append only files that always write data to the end of the file.

These files allow column databases to avoid waiting for the definitive writes on persistent supports.

Whenever a write operation is sent to the database, is stored into a commit log. Then, the data is updated into the specific server and disk block.

One way to ensure this is to have the database write data to disk (or other persistent storage) before sending the success response. The database could do this, but it would have to wait for the read/write heads to be in the correct position on the disk before writing the data. If the database did this for every write, it could significantly cut down on write performance.

Instead of writing data immediately to its partition and disk block, column family databases can employ commit logs. These are append only files that always write data to the end of the file.

In the event of a failure, the database management system reads the commit log on recovery. Any entries in the commit log that have not been saved to partitions are then written to appropriate partitions.

After a failure, the **recovery** process reads the commit logs and writes entries to partitions.

BLOOM FILTERS

Bloom filters are **probabilistic** data structures.

Bloom filters help to **reduce** the number of **reading operations**, avoiding reading partitions that certainly do not contain a specific searched data.

A Bloom filter tests whether or not an element is a member of a set. Unlike a typical member function, the Bloom filter sometimes returns an incorrect answer. It could return a positive response in cases where the tested element is not a member of the set. This is known as a **false-positive**. Bloom filters **never return a negative response** unless the element is not in the set.

Member Function

Input Set	Test Element	In Set
{a,b,c}	a	Yes
{a,b,c}	c	Yes
{a,b,c}	e	No

Bloom Filter

Input Set	Test Element	In Set
{a,b,c}	a	Yes
{a,b,c}	c	Yes
{a,b,c}	e	Yes
{a,b,c}	f	No
{a,b,c}	g	No

Low Probability but Possible

Another way to achieve the same benefit is to use a hash function. For example, assume you partition customer data using a hash function on a person's last name and city. The hash function would return a single value for each last name–city combination. The application would only need to read that one partition. Why should database developers use Bloom filters that sometimes return incorrect?

Both HBase and Cassandra make use of Bloom filters to avoid unnecessary disk seeks.

EXAMPLE

Let consider a specific partition. A bloom filter for this partition can be an array of n binary values. At the beginning, all the elements are set to 0.

Let consider also a set of k << n hash functions which map each data to be stored into the partition in an element of the array.

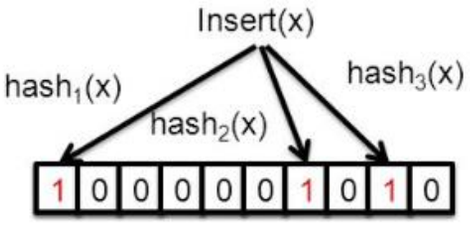
When a new data is added (write) to the partition, all the hash functions are calculated and the corresponding bit are set to one.

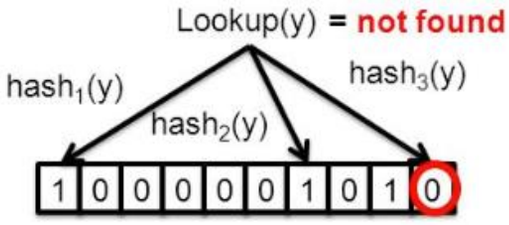
When a read operation is required, all the has functions are calculated and the access to the partition is actually performed if and only if all the corresponding element in the array are set to 1.

The maximum false positive rate depends on the values of n and k.

- Let consider a Bloom Filter with n = 10 and k = 3.
- The Insert function describes a write operation
- The Lookup function describes a read operation

p false negatives decrease when n increase





CONSISTENCY LEVEL

Consistency level refers to the consistency **between copies** of data on **different replicas**. In the strictest sense, **data is consistent only if all replicas have the same data**. At the other end of the spectrum, you could consider the data "consistent" as long as it is persistently written to at least one replica. There are several intermediate levels as well.

Consistency level is set according to several, sometimes competing, requirements:

- How fast should write operations return a success status after saving data to persistent storage?
- Is it acceptable for two users to look up a set of columns by the same row ID and receive different data?
- If your application runs across multiple data centers and one of the data centers fails, must the remaining functioning data centers have the latest data?
- Can you tolerate some inconsistency in reads but need updates saved to two or more replicas?

In many cases, a **low consistency level** (the data must be written in at least one replica) can satisfy requirements. Consider an application that collects sensor data every minute from hundreds of industrial sensors. If data is occasionally lost, the data sets will have missing values.

To avoid disrupting players' games in the event of a server failure, an underlying column family database could be configured with a consistency level requiring writes to two or three replicas. Using a **higher level of consistency** (all replicas must have the same data) would increase availability but at the cost of slowing write operations and possibly adversely affecting gameplay. The highest levels of consistency, such as writing replicas to multiple replicas in multiple data centers, should be saved for the most demanding fault-tolerant applications.

Other situations call for a **moderate consistency level** (all the intermediate solutions). Players using an online game reasonably expect to have the state of their game saved when they pause or stop playing on one device to switch to another. Even losing a small amount of data could frustrate users who have to repeat play and possibly lose gains made in the game.

REPLICATION

Replication is a **process closely related to consistency level**. Whereas the consistency level determines how many replicas to keep, the replication process determines where to **place replicas and how to keep them up to date**. Each database has its own replication strategies.

In the simplest case, the server for the first replica is determined by hash function, and additional replicas are placed according to the relative position of other servers. For example, all nodes in Cassandra are in a logical ring. Once the first replica is placed, additional replicas are stored on successive nodes in the ring in the clockwise direction.

Column family databases can also use network topology to determine where to place replicas. For example, replicas may be created on different racks within a data center to ensure availability in the event of a rack failure.

ANTI-ENTROPY

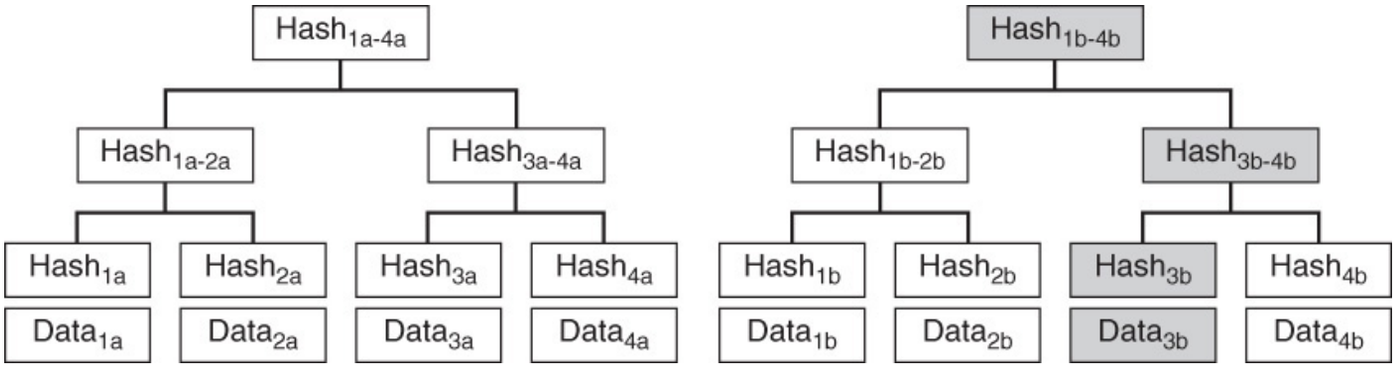
Anti-entropy is the process of detecting differences in replicas. It is important to detect and resolve inconsistencies among replicas exchanging a low amount of data. Column family databases can exploit the fact that much of replica data may not change between anti-entropy checks. They do this by sending hashes of data instead of the data itself.

> The second law of thermodynamics describes a feature of entropy, which is the state of randomness and lack of order in a system or object. A broken glass, for example, has higher entropy than an unbroken glass. The second law of thermodynamics states that the amount of entropy (or disorder) in a closed system does not decrease. A broken glass does not repair itself and restore itself to the state of less entropy found in an unbroken glass.
> Databases, especially distributed databases, are subject to a kind of entropy, too. The mechanical parts of a database server are certainly subject to entropy—just ask anyone who has suffered a disk failure—but that is not the kind of entropy discussed here.
> Distributed database designers have to address information entropy. Information entropy increases when data is inconsistent in the database. If one replica of data indicates that Lucinda Jones last made a purchase on January 15, 2014, and another replica has data that indicates she last made a purchase on November 23, 2014, the system is in an inconsistent state.

One method employs a **tree of hashes**. The leaf nodes contain hashes of a data set. The nodes above the leaf nodes contain a hash of the hashes in the leaf nodes. Each successive layer contains the hash of hashes in the level below. The root node contains the hash of the entire collection of data sets.

Anti-entropy processes can calculate hash trees on all replicas. One replica sends its hash tree to another node. That node compares the hash values in the two root nodes. If both are the same, then there is no difference in the replicas. If there is a difference, then the antientropy process can compare the hash values at the next level down.

At least one pair of these hash values will differ between replicas. The process of traversing the tree continues until the process reaches one or more leaf nodes that have changed. Only the data associated with those leaf nodes needs to be exchanged.



Hash trees, or Merkle trees, allow for rapid checks on consistency between two data sets. In this example, data3a and data3b are different, resulting in different hash values in each level from the data block to the root.

GOSSIP PROTOCOL – COMMUNICATION PROTOCOLS

In distributed systems, each node needs to be aware about the status of the other nodes.

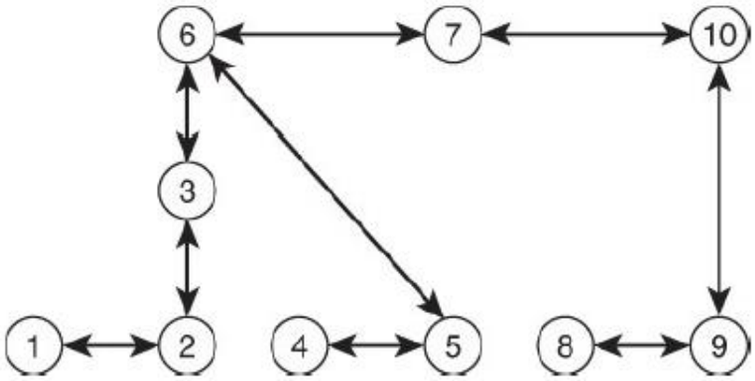
This is not too much of a problem when the number of nodes in a cluster is small, say fewer than 10 servers. If every node in a server has to communicate with every other node, then the number of messages can quickly grow.

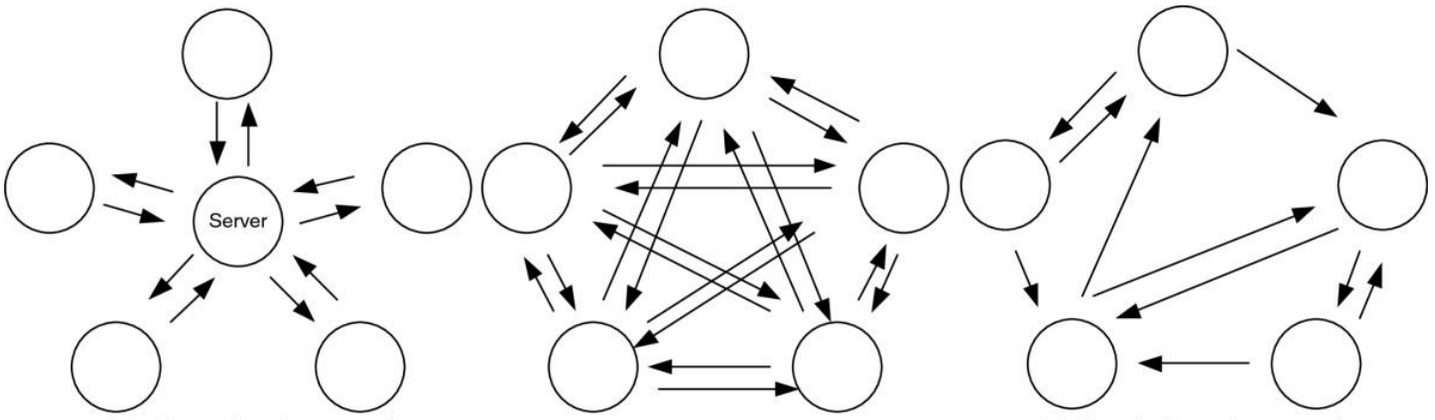
The number of total messages to exchange for a complete communication is equal to $n * \frac{n-1}{2}$, where n is the number of nodes of a cluster.

Number of Nodes	Number of Messages for Complete Communication
2	2
3	6
4	12
5	20
10	90
15	210
20	380
25	600
50	2,450
100	9,900

Instead of having every node communicate with every other node, it is more efficient to have nodes share information about themselves as well as other nodes from which they have received updates. Consider a cluster with 10 nodes. If each node communicated with every other node, the system will send a total of 90 messages to ensure all nodes have up-to-date information.

When using a gossip protocol—in which each node sends information about itself and all information it has received from other nodes—all nodes can be updated with a fraction of the number of messages required for complete communication.





(a) Centralized approach, which uses a centralized server for forwarding and mixing of multimedia streams

(b) Fully connected overlay approach, where all peers are in direct contact with one another

(c) Gossip-based approach, where peers operate in parallel, and each peer communicates with one or more randomly selected partner

HINTED HANDOFF

Replicas enable read availability even if some nodes have failed. They do not address how to handle a write operation that is directed to a node that is down. The hinted handoff mechanism is designed to solve this problem.

Two scenarios may happen:

1) If a write operation is directed to a node that is unavailable, the operation can be redirected to another node, such as another replica node or a node designated to receive write operations when the target node is down.
2) The node receiving the redirected write message **creates** a data structure to store information about the write operation and where it should ultimately be sent. The hinted handoff **periodically checks** the status of the target server and **sends** the write operation when the target is available.

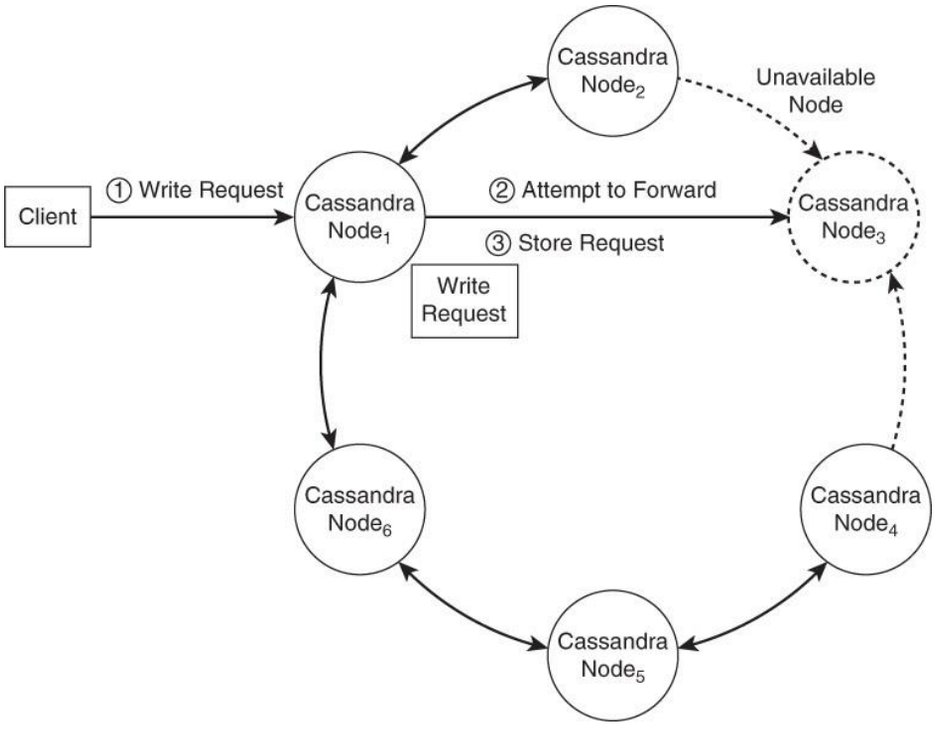
Once the write data is successfully written to the target node, it can be considered a successful write for the purposes of consistency and replication.

EXAMPLE

Cassandra is known for being well suited for write-intensive applications. This is probably due in part to its ability to keep accepting write requests even when the server that is responsible for handling the write request is unavailable.

The client sends a request to Node 1 to write data associated with a row key. Node 1 queries its local copy of metadata about the cluster and determines that Node 3 should process this request. Node 1, however, knows that Node 3 is unavailable because the gossip protocol informed Node 1 about the status of Node 3 a few seconds ago. Rather than lose the write information or change the permanent location of data associated with that row key, Node 1 initiates a hinted handoff.

A hinted handoff entails storing information about the write operation on a proxy node and periodically checking the status of the unavailable node. When that node becomes available again, the node with the write information sends, or "hands off," the write request to the recently recovered node.



WHEN TO USE COLUMN FAMILY DATABASES

Column family databases are appropriate choices for large-scale database deployments that require **high levels of write performance**, a **large number of servers** or multi–data center **availability**.

Cassandra's peer-to-peer architecture with support for hinted handoff means the database will always be able to accept write operations as long as at least one node is functioning and reachable. Write-intensive operations, such as those found in social networking applications, are good candidates for using column family databases.

If your write-intensive application also requires transactions, then a column family database may not be the best choice. You might want to consider a hybrid approach that uses a database that supports ACID transactions (for example, a relational database or a key-value database such as FoundationDB5).

Column family databases are also appropriate when a large number of servers are required to meet expected workloads. Column family databases typically run with more than several servers. If you find one or a few servers satisfy your performance requirements, you might find that key-value, document, or even relational databases are a better option.

Cassandra supports multi–data center deployment, including multi–data center replication. If you require continuous availability even in the event of a data center outage, then consider Cassandra for your deployment.

If you are considering column family databases for the flexibility of the data model, be sure to evaluate key-value and document databases. You may find they meet your requirements and run well in an environment with a single server or a small number of servers.

DESIGNING FOR COLUMN FAMILY DATABASES

In the following, we show some examples of typical queries that may led to choose column DB:

- How many new orders were placed in the Northwest region yesterday?
- When did a particular customer last place an order?
- What orders are en route to customers in London, England?
- What products in the Ohio warehouse have fewer than the stock keeping minimum number of items?

Only when you have questions like these can you design for a column family database. Like other NoSQL databases, design starts with queries.

Queries provide information needed to effectively design column family databases. The information includes

- Entities
- Attributes of entities
- Query criteria
- Derived values

Entities represent things that can range from concrete things, such as customers and products, to abstractions such as a service level agreement or a credit score history. Entities are modeled as rows in column family databases.

Designers start with this information and then use the **features of column-based** databases management systems to select the most appropriate **implementation**.

HOW TO USE THE EXTRACTED INFORMATION

- Entities
  - A single row describe the instance of to a single entity.
  - Rows are uniquely identified by row keys.
- Attributes
  - Attributes of entities are modeled using columns.
- Query criteria
  - The selection criteria should be used to determine optimal ways to organize data with tables and column families and how to build partitions
    - Queries that require range scans, such as selecting all orders placed between two dates, are best served by tables that order the data in the same order it will be scanned.

- Use the set of attributes to return to help determine how to group attributes (most efficient to store columns together that are frequently used together)

➢ Derived values
- it is an indication that additional attributes may be needed to store derived data.
  - such as a count of orders placed yesterday or the average dollar value of an order

DIFFERENCES WITH RELATIONAL DBS

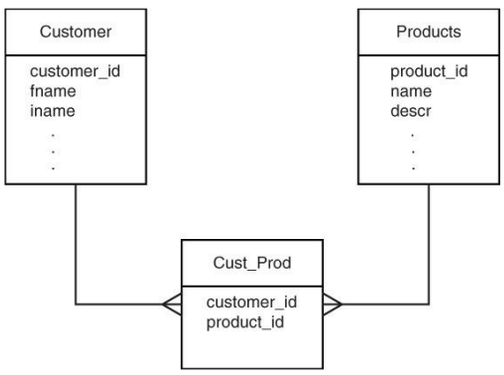
Column family databases are implemented differently than relational databases. Thinking they are essentially the same could lead to poor design decisions. It is important to understand:

- Column family databases are implemented as sparse, multidimensional maps.
- Columns can vary between rows.
- Columns can be added dynamically.
- Joins are not used because data is denormalized.
- It is suggested to use separate keyspace for each application.

DENORMALIZE INSTEAD OF JOIN

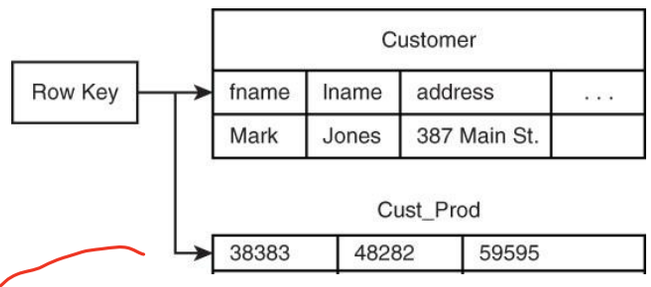
Tables model entities, so it is reasonable to expect to have one table per entity. Column family databases often need fewer tables than their relational counterparts.

MANY-TO-MANY RELATIONSHIP



This is because column family databases denormalize data to avoid the need for joins. For example, in a relational database, you typically use three tables to represent a many-to-many relationship: two tables for the related entities and one table for the many-to-many relation.

While, in the case of denormalized data, each customer includes a set of column names that correspond to purchased products. Similarly, products include a list of customer IDs that indicate the set of customers that purchased those products.



In order to avoid join operations, each customer includes a set of column names that correspond to purchased products (Ids).

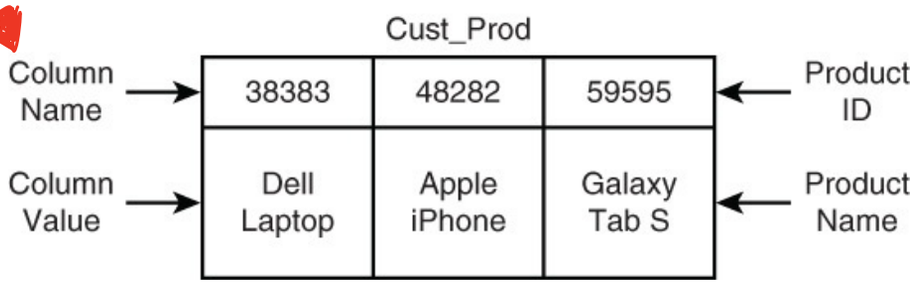
The features of the product, such as description, size, color, and weight, are stored in the products table. If your application users want to produce a report listing products bought by a customer, they probably want the product name in addition to its identifier. Because you are dealing with large volumes of data (otherwise

Evitiamo i Join salvando le info che => Non ci interessa se lo spazio necessario

ci servono nel customer aumenta, vogliamo migliorare le write

you would not be using a column family database), you do not want to join or query both the customer and the product table to produce the report.

SOLUTION: the customer table includes a list of column names indicating the product ID of items purchased by the customer. Because the column value is not used for anything else, you can store the product name there.

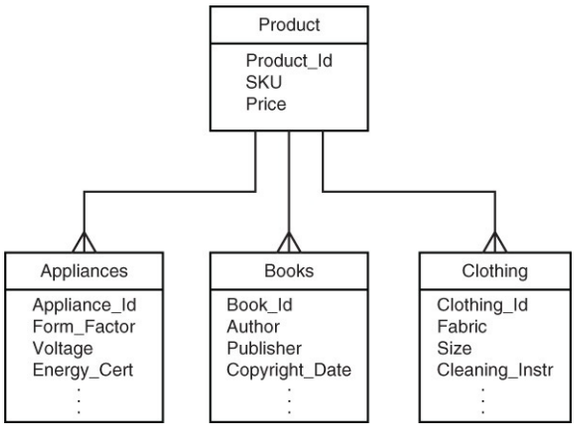


Keeping a copy of the product name in the customer table will increase the amount of storage used (we improve the read performance).

MODEL AN ENTITY WITH A SINGLE ROW

A single entity, such as a particular customer or a specific product, should have all its attributes in a single row. This can lead to cases in which some rows store more column values than others, but that is not uncommon in column family databases.

The retailer designing the application sells several different types of products, including appliances, books, and clothing. They all share some common attributes, such as price, stock keeping unit (SKU), and inventory level. They each have unique features as well. One way to model this is with several tables:



In this case we have the classical organization in a relational DB of an entity that can assume different “shapes”.

Column family databases do not provide the same level of transaction control as relational databases. Typically, writes to a row are atomic. If you update several columns in a table, they will all be updated, or none of them will be.

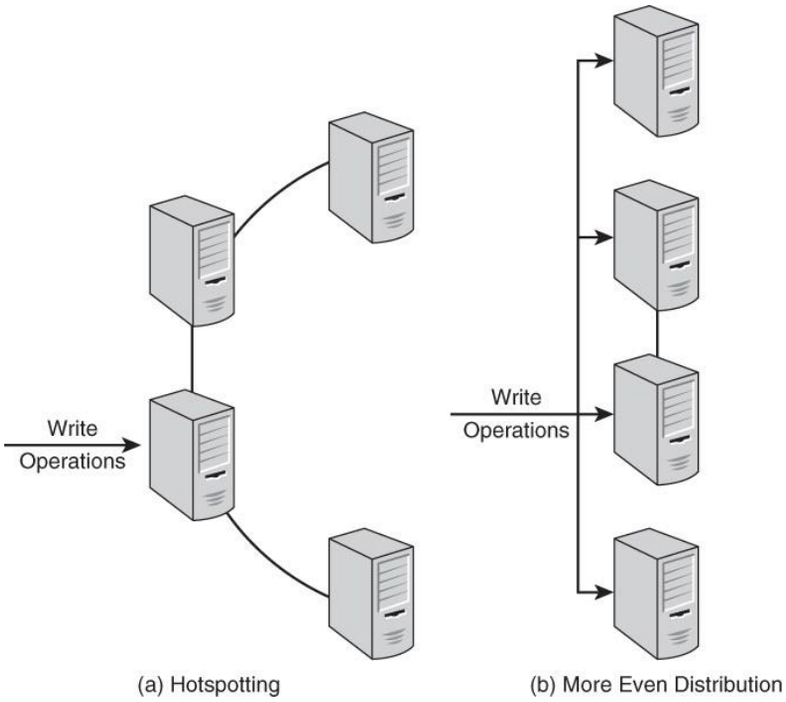
Caution: If you need to update two separate tables, such as a product table and a books table, it is conceivable the updates to the product table succeed, but the updates to the book table do not. In such a case, you would be left with inconsistent data.

In order to ensure the **atomicity** of the operation in column DB, the same entity than in the figure must be organized in just **one table with 4 column families**.

AVOID HOTSPOTTING IN ROW KEYS

Distributed systems enable you to take advantage of large numbers of servers to solve problems. It is inefficient to direct an excessive amount of work at one or a few machines while others are underutilized.

Hotspotting occurs when many operations are performed on a small number of servers



(a) Hotspotting (b) More Even Distribution

We can prevent hotspotting by hashing sequential values generated by other systems.

Alternatively, we could add a random string as a prefix to the sequential value.

These strategies would eliminate the effects of the lexicographic order of the source file on the data load process.

Consider an example of how this can occur in HBase. HBase uses lexigraphic ordering of rows. Let's assume you are loading data into a table and the key value for the table is a sequential number assigned by a source system. The data is stored in a file in sequential order. As HBase loads each record, it will likely write it to the same server that received the prior record and to a data block near the data block of the prior record. This helps avoid disk latency, but it means a single server is working consistently while others are underutilized.

You can prevent hotspotting by hashing sequential values generated by other systems. Alternatively, you could add a random string as a prefix to the sequential value. This would eliminate the effects of the lexicographic order of the source file on the data load process.

KEEP AN APPROPRIATE NUMBER OF COLUMN VALUE VERSIONS

HBase enables you to store multiple versions of a column value. Column values are timestamped so you can determine the latest and earliest values. Like other forms of version control, 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.

HBase enables you to set a minimum and maximum number of versions.

It will not remove versions if it would leave a column value with less than the minimum number of versions. When the number of versions exceeds the maximum number of versions, the oldest versions are removed during data compaction operations.

AVOID COMPLEX DATA STRUCTURES IN COLUMN VALUES

Any kind of data structure may be stored in a column value such a JSON file (may contain an embedded document):

```
{
    "customer_id":187693,
    "name": "Kiera Brown",
    "address" : {
            "street" : "1232 Sandy Blvd.",
            "city" :  "Vancouver",
            "state" :  "Washington",
            "zip" :  "99121"
               },
   "first_order" : "01/15/2013",
   "last_order" : " 06/27/2014"
}
```

This type of data structure may be stored in a column value, but it is not recommended unless there is a specific reason to maintain this structure. If you are simply treating this object as a string and will only store and fetch it, then it is reasonable to store the string as is. If you expect to use the database 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 apply database features to the attributes. For example, creating separate columns for street, city, state, and zip means you can create secondary indexes on those values (**indexing**).

Also, separating attributes into individual columns allows you to use different column families if needed. Both the ability to use secondary indexes and the option of separating columns according to how they are used can lead to improved database performance.

One of the most important considerations with regard to performance is indexing.

INDEXING: PRIMARY AND SECONDARY INDEXES

Index is a specific data structure in a DBMS that allows the database engine to retrieve data faster than it otherwise would. In column databases, we can look up a column value to quickly find rows that reference that column value. For example, if you want to look up customers in a particular state:

```
SELECT
    fname, lname
FROM
    customers
WHERE
     state = 'OR';
```

A database index functions much like the index in a book. You can look up an entry in a book index to find the pages that reference that word or term. Similarly, in column family databases, you can look up a column value, such as state abbreviation, to find rows that reference that column value. In many cases, using an index allows the database engine to retrieve data faster than it otherwise would.

It is helpful to distinguish two kinds of indexes: **primary** and **secondary**.

- Primary indexes are indexes on the row keys of a table
  - They are automatically maintained by the column family database system
- Secondary indexes are indexes created on one or more column values.
  - Either the database system or your application can create and manage secondary indexes.

SECONDARY INDEXES MANAGED BY THE DATABASE MANAGEMENT SYSTEM

General rule: if you need secondary indexes on column values and the column family database system provides automatically managed secondary indexes, then you should use them.

Cassandra will then create and manage all data structures needed to maintain the index. It will also determine the optimal use of indexes. For example, if you have an index on state and last name column values and you queried the following, Cassandra would choose which index to use first:

```
SELECT
    fname, lname
FROM
    customers
WHERE
    state = 'OR'
AND
    lname = 'Smith'
```

we can speedup this query

By setting an index on the state and on the lname attributes.

Typically, the DBMS will use the **most selective** index first.

For example, if there are 10,000 customers in Oregon and 1,500 customers with the last name Smith, then it would use the lname secondary index first. It might then use the state index to determine, which, if any, of the 1,500 customers with the last name Smith are in Oregon.

The automatic use of secondary indexes has another major advantage because you do not have to change your code to use the indexes.

WHEN AVOIDING TO USE AUTOMATICALLY MANAGED INDEXES

There are times when you should not use automatically managed indexes. Avoid, or at least carefully test, the use of indexes in the following cases:

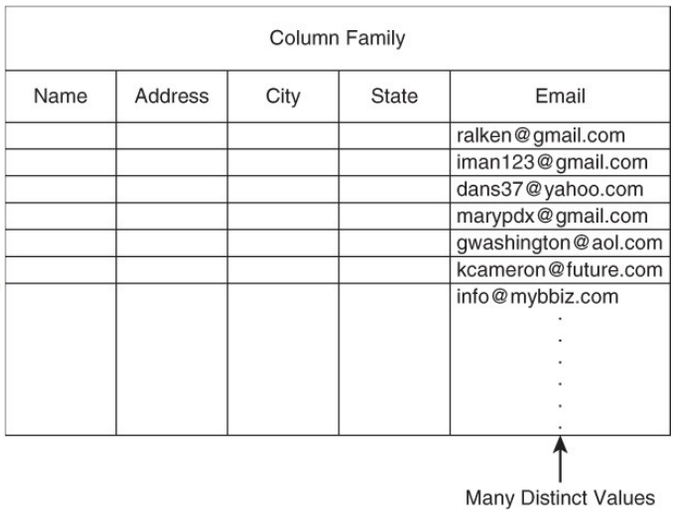
- There is a small number of distinct values in a column.
- There are many unique values in a column.
- The column values are sparse.

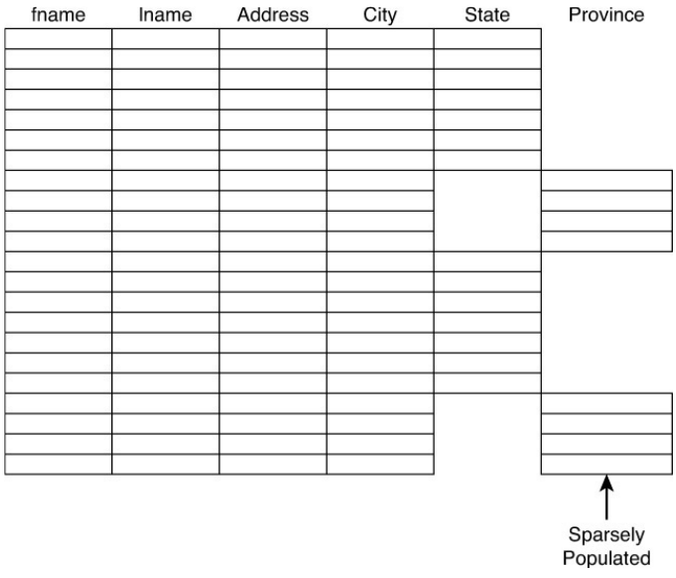
Column Family				
Name	Address	Opt In?	City	. . . . .
		Y		
		Y		
		N		
		Y		
		N		
		. . .		
		Y		
		N		

Only Two Distinct Values

When the number of distinct values in a column (known as the cardinality of the column) is **small**, indexes will **not help performance** much—it might even hurt. For example, if you have a column with values Yes and No, an index will probably not help much, especially if there are roughly equal numbers of each value.

At the other end of the cardinality spectrum are columns with **many distinct values**, such as street addresses and email address. Again, automatically managed indexes may not help much here because the index will have to maintain so much data it could take more time to search the index and retrieve the data than to scan the tables for the particular value.





In cases where many of the rows do not use a column, a secondary index may not help. For example, if most of your customers are in the United States, then their addresses will include a value in the state column. For those customers who live in Canada, they will have values in the province column instead of in the state column. Because most of your customers are in the United States, the province column will have sparse data. An index will not likely help with that column.

CREATE AND MANAGE SECONDARY INDEXES USING TABLES

If your column family database system does not support automatically managed secondary indexes or the column you would like to index has many distinct values, you might benefit from creating and managing your own indexes.

Indexes created and managed by your application use the same table, column family, and column data structures used to store your data. Instead of using a statement such as CREATE INDEX to make data structures managed by the database system, you explicitly create tables to store data you would like to access via the index.

Let's return to the customer and product database. Your end users would like to generate reports that list all customers who bought a particular product. They would also like a report on particular products and which customers bought them.

In the first situation, you would want to quickly find information about a product, such as its name and description. The existing product table meets this requirement. Next, you would want to quickly find all customers who bought that product. A time-efficient way to do this is to keep a table that uses the product identifier as the row key and uses customer identifiers as column names. The column values can be used to store additional information about the customers, such as their names. The necessary data is stored in the Cust _ by _ Prod table.

Customer

Row key	fname	lname	street	city	state
123	Jane	Smith	387 Main St	Boise	ID
287	Mark	Jones	192 Wellfleet Dr	Austin	TX
1987	Harsha	Badal	298 Commercial St	Provincetown	MA
2405	Senica	Washington	98 Morton Ave	Windsor	CT
3902	Marg	O'Malley	981 Circle Dr	Santa Fe	NM

Product

Row key	name	descr	qty_avail	category	
38383	Dell Latitude E6410	Laptop with ...	124	Computer	
48282	Apple iPhone	iPhone 6 with ...	345	Phone	
59595	Galaxy Tab S	Samsung tablet ...	743	Tablet	

Cust_by_Prod

Row key	123	287	1987	2405	3902
38383	Smith		Badal		
48282	Smith	Jones			O'Malley
59595				Washington	

Prod_by_Cust

Row key	38383	48282	59595		
123	Dell Latitude E6410	Apple iPhone			
287		Apple iPhone			
1987	Dell Latitude E6410				
2405			Galaxy Tab S		
3902		Apple iPhone			

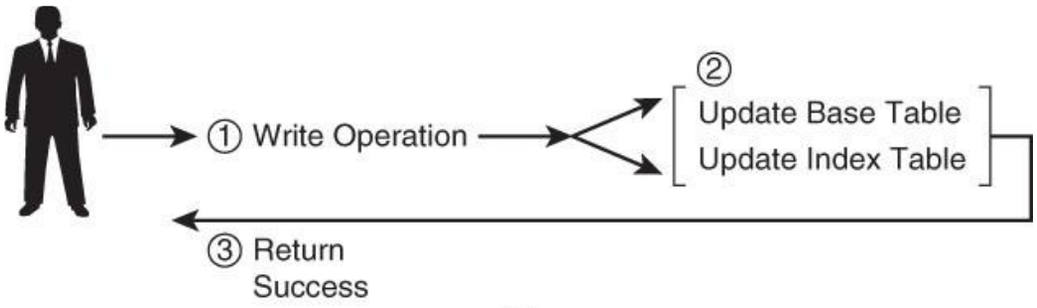
A similar approach works for the second report as well. To list all products purchased by a customer, you start with the customer table to find the customer identifier and any information about the customer needed for the report, for example, address, credit score, last purchase date, and so forth. Information about the products purchased is found in the Prod _ by _ Cust.

Using tables as secondary indexes will, of course, require more storage than if no additional tables were used. The same is the case when using column family database systems to manage indexes. In both cases, you are trading additional storage space for better performance.

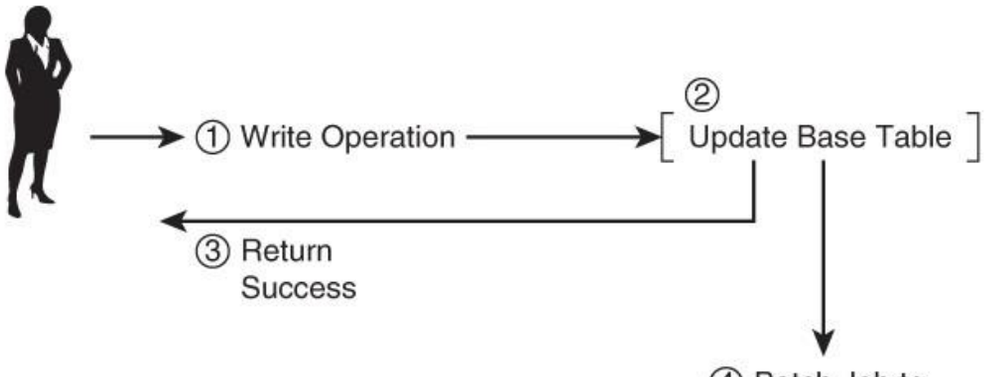
When using tables as indexes, you will be responsible for maintaining the indexes. You have two broad options with regard to the timing of updates. You could update the index whenever there is a change to the base tables, for example, when a customer makes a purchase. Alternatively, you could run a batch job at regular intervals to update the index tables.

Updating index tables at the same time you update the base tables keeps the indexes up to date at all times. This is a significant advantage if the reports that use the index table could be run at any time. A drawback of this approach is that your application will have to perform two write operations, one to the base table and one to the index table. This could lead to longer latencies during write operations.

Updating index tables with batch jobs has the advantage of not adding additional work to write operations. The obvious disadvantage is that there is a period of time when the data in the base tables and the indexes is out of synchronization. This might be acceptable in some cases. For example, if the reports that use the index tables only run at night as part of a larger batch job, then the index tables could be updated just prior to running the report. Your reporting requirements should guide your choice of update strategy.



(a)



(b)

(a) Updating an index table during write operations keeps data synchronized but increases the time needed to complete a write operation. (b) Batch updates introduce periods of time when the data is not synchronized, but this may be acceptable in some cases.

GRAPH DATABASE

TERMINOLOGY – ELEMENTS OF A GRAPH

VERTEX

↗ Come se fosse una primary key

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:

- 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
- Compute servers in a cluster

Vertices can have properties:

- in a social network represents a person
    - name, an address, and a birth date
- highway system uses vertices to represent cities
    - populations, a longitude and latitude, and a name, and are located in a geographic region.

EDGE

An edge (link or arc) **defines relationships between vertices or objects** connecting vertices.

For example, in a family tree database, vertices can represent people, whereas the edges represent the relationships between them, such as “daughter of” and “father of.” In the case of a highway database, cities are represented with vertices, whereas edges represent highways linking the cities.

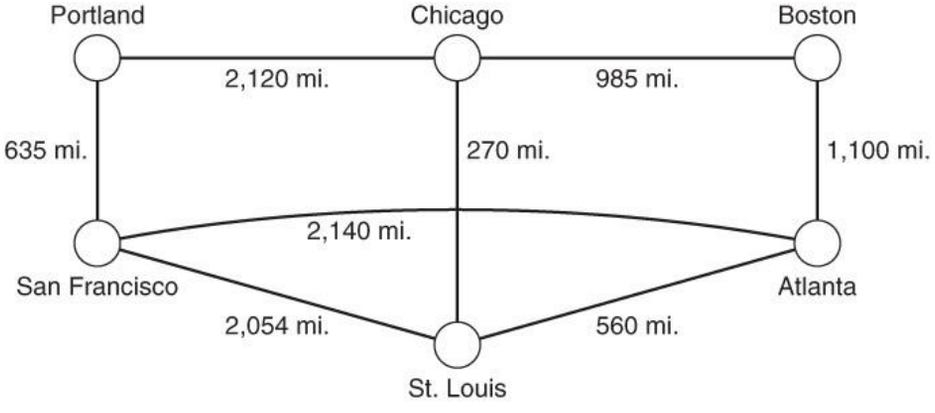
Also edges have properties:

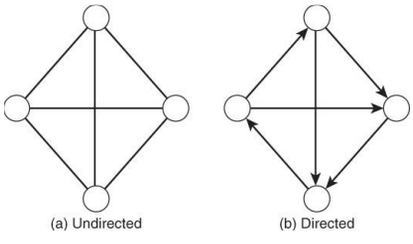
- in the highway database
    - distance, speed limit, and number of lanes.
- In the family tree
    - indicating whether two people are related by marriage, adoption, or biology.

A commonly used property is called the **weight of an edge**. Weights represent some value about the relationship:

- In highways
    - distance between cities
- In social network
    - how frequently the two individuals post on each other’s walls or comment on each other’s posts

In general, weights can represent cost, distance, or another measure of the relationship between objects represented by vertices.

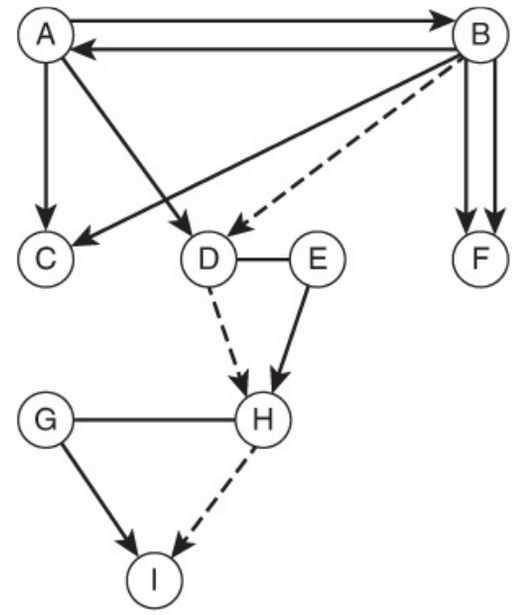




There are two types of edges: **directed and undirected**. Directed edges have a direction. This is used to indicate how the relationship, as modeled by the edge, should be interpreted. For example, in a family relations graph, there is a direction associated with a “parent of” relation. However, direction is not always needed. The highway graph, for instance, could be undirected, assuming traffic flows both ways.

PATH

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.



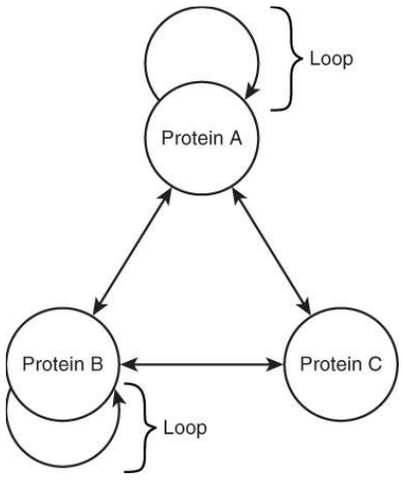
I → H → D → B Ancestor Path

Paths are important because they capture information about how vertices in a graph are related. For example, in a family graph, a person is an ancestor of someone else only if there is a directed path from the person to her ancestor. In the case of a family tree, there is only one path from a person to an ancestor. In the case of a highway graph, there may be multiple paths between cities.

A common problem encountered when working with graphs is to find the least weighted path between two vertices. The weight can represent cost of using the edge, time required to traverse the edge, or some other metric that you are trying to minimize.

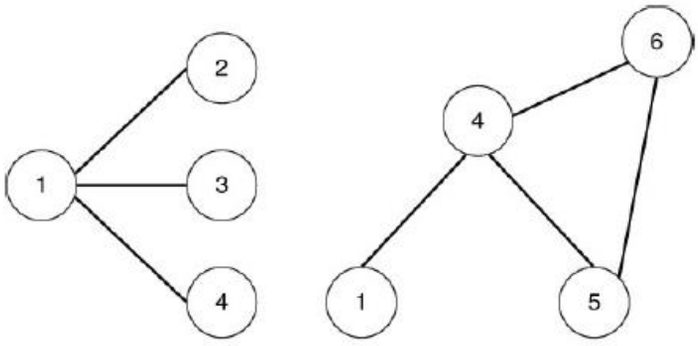
LOOP

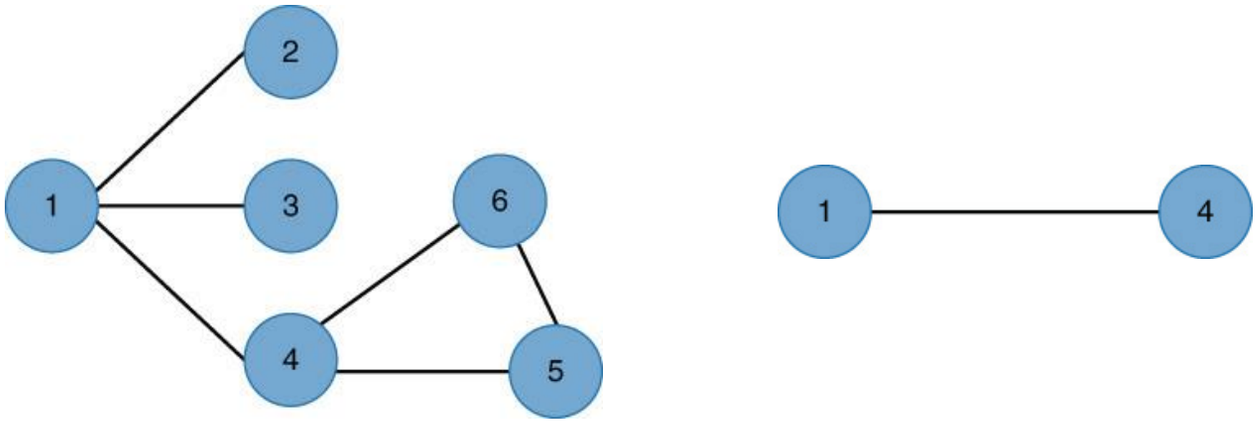
A loop is an edge that connects a vertex to itself. For example, in biology, proteins can interact with other proteins. Some proteins interact with other protein molecules of the same type. A loop could be used to represent this. However, like direction, it might not make sense to allow loops in some graphs. For instance, a loop would not make much sense in a family tree graph; people cannot be their own parents or children.



OPERATIONS ON GRAPHS

The union of graphs is the combined set of vertices and edges in a graph. Consider two graphs. The first graph, A, has vertices 1, 2, 3, and 4, and the edges are {1, 2}, {1, 3}, and {1, 4}. The second graph, B, has vertices 1, 4, 5, and 6, and edges {1, 4}, {4, 5}, {4, 6}, and {5, 6}. Figure 13.7 shows the two graphs.



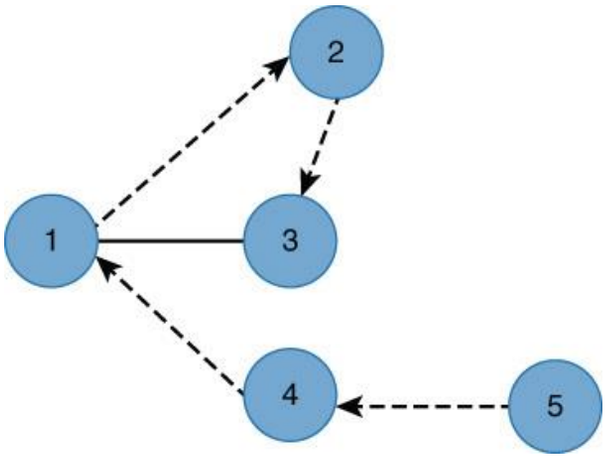


The **union** of A and B is the set of vertices and edges from both graphs. The set of vertices is 1, 2, 3, 4, 5, and 6. The set of edges is {1, 2}, {1, 3}, {1, 4}, {4, 5}, {4, 6}, and {5, 6}.

The **intersection** of a graph is the set of vertices and edges that are common to both graphs. In the case of graphs A and B, the intersection of graphs includes vertices 1 and 4, as well as the edge {1, 4}.

GRAPH TRAVERSAL

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.



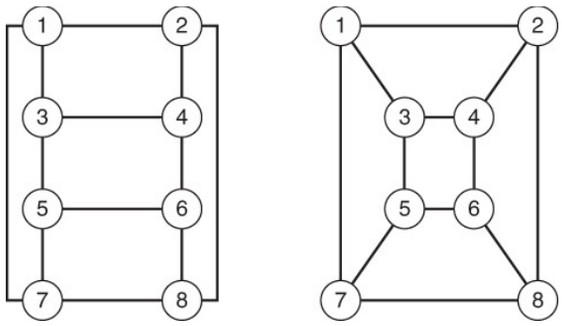
For example, you might create a graph of all cities in the country that you would like to visit; cities are represented by vertices and highways by edges. You start at your home city and follow the highway with the shortest distance out of all edges between your starting city and another city on the graph.

After you visit the next city, you drive on to a third city. The third city is the city that is the shortest distance from the current city, unless you have already been to that city. For instance, you have already been to your home city, so even if your home city is closest to the second city, you would choose the next shortest route to an adjacent city. In this way, you could keep moving from city to city until you have visited all cities.

The traversal of the graph is often carried out in a particular way, such us **minimizing the cost** of the path.

ISOMORPHISM

Two graphs are considered isomorphic if for each vertex in the first graph, there is a corresponding vertex in the other graph. In addition, for each edge between a pair of vertices in the first graph, there is a corresponding edge between the corresponding vertices of the other graph.



- For each vertex in the first graph, there is a corresponding vertex in the other graph
- For each edge between a pair of vertices in the first graph, there is a corresponding edge between the corresponding vertices of the other graph.

Graph isomorphism is important if you are trying to detect patterns in a set of graphs. In a large social network graph, there may be repeating patterns with interesting properties. For example, it may be possible to detect business collaborators by examining their links on a business social network.

Another branch of study that makes use of graphs is epidemiology, or the study of infectious diseases. For example, an epidemiologist who studies flu transmission in a city might build a graph of individuals and their connections to other individuals. Let's assume that they can collect data about who has the flu at any point in time and they want to determine how fast it spreads.

First, the flu may spread faster in some groups than others. This may be because of the characteristics of the individuals involved, or it could be because of patterns of interconnection that affect the rate of disease spread. If epidemiologists can identify patterns associated with infection, they could then identify other individuals by finding similar patterns and target them for intervention, education, and so on.

Detecting isomorphism in graphs or sub graphs allows us to identify **useful patterns**.

ORDER AND SIZE

Order and size are measures of **how large a graph is**.

- **order** of a graph: the number of vertices
- **size** of a graph: the number of edges in a graph

The order and size of a graph are important to understand because they can affect the time and space required to perform operations. It is obvious that performing a union or intersection on a small graph would take less time than performing the same operation on a larger graph. It is also easy to assume that traversing a small graph will take less time than traversing a large graph.

Some problems sound simple but can quickly become too hard to solve in any reasonable amount of time. Consider a clique, which is a set of vertices in a graph that are all connected to each other. Finding cliques is impractical for large graphs.

Think of trying to find the largest subset of people in a social network that know each other; this is obviously a large undertaking. As you work with graphs and perform operations on graphs, consider how the order and size impact the time it takes to perform operations. The higher the order and the size of the graph, the harder will be to solve the query!

DEGREE

Degree is the number of edges linked to a vertex and is one way to **measure the importance of any given vertex in a graph**. Vertices with high degrees are more directly connected to other vertices than vertices with low degrees. Degree is important when addressing problems of spreading information or properties through a network.

Consider a person with many family members and friends that he sees regularly; that person would have high degree. What if that person contracts the flu? It is easy to imagine it spreading to friends and family, and from there to people outside of the initial social circle. One person can infect many people if he has many connections.

As another example, think about the last time you were delayed in an airport because of bad weather. Delays in airports with high degrees, like Chicago and Atlanta, can generate ripple effects that lead to delays at other airports.

CLOSENESS

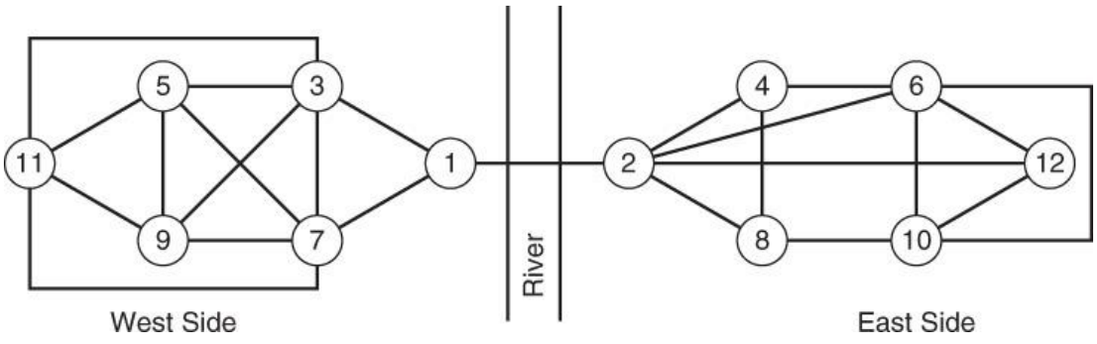
Closeness is a property of a vertex that indicates **how far the vertex is from all others in the graph**.

Closeness is an important measure if you want to understand the spread of information in a social network, an infectious disease in a community, or movement of materials in a distribution network.

Vertices with high closeness values can reach other vertices in the network faster than vertices with smaller closeness values. Marketers, for example, might want to target people in a social network with high closeness values to get the word out about a new product. Information will spread faster in the network if the marketer starts with someone with a high closeness value than with someone on the periphery of the network. It is useful to evaluate the speed of information spreading in a network, starting from a specific node.

BETWEENNESS

In addition to understanding closeness, it is sometimes important to understand betweenness. Betweenness is a measure of **how much of a bottleneck a given vertex is**. Imagine a city on a river that has many roads but only one bridge



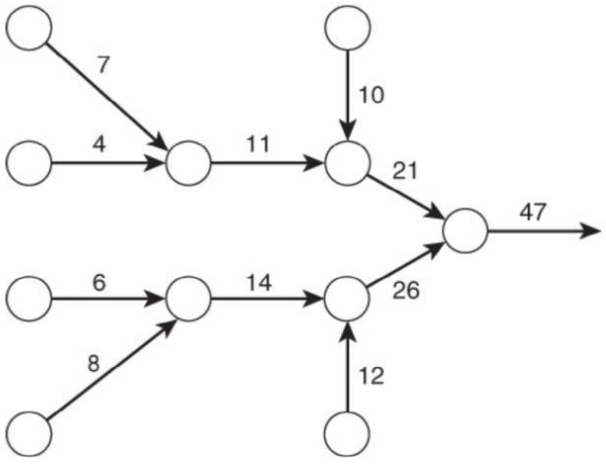
As you can see from the network, there are many ways, or paths, to move from one vertex to another on the west side of the network. Similarly, there are multiple paths to get from one vertex to another on the east side. There is only one edge that connects the west and east sides of the city, linking vertices 1 and 2.

Both vertices 1 and 2 will have high betweenness scores as they form a bottleneck in the graph. If vertex 1 or 2 were removed, it would leave the graph disconnected. However, if you removed nodes 4 or 9, for example, you could still move between any of the remaining nodes.

Betweenness helps identify potentially vulnerable parts of a network. For instance, you would not want a distribution network that depended on one bridge. If that bridge were damaged or the flow of traffic were disrupted, you would not be able to move materials to all vertices in the network.

FLOW NETWORK

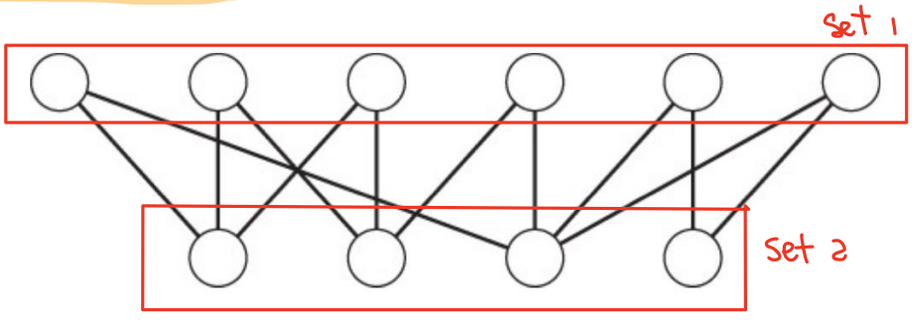
A flow network is a **directed graph** in which **each edge has a capacity** and each vertex has a set of incoming and outgoing edges. The sum of the capacity of incoming edges cannot be greater than the sum of the capacity of outgoing edges. The **two exceptions to this rule** are **source and sink vertices**. Sources have no inputs but do have outputs, whereas sinks have inputs but no outputs.



Flow networks are also called transportation networks. Graph databases can be used to model flow networks, like road systems or transportation networks. They can also be used to model processes with continuous flows, such as a network of storm drains that take in rainwater (source) and allow it to flow into a river (sink).

BIPARTITE GRAPH

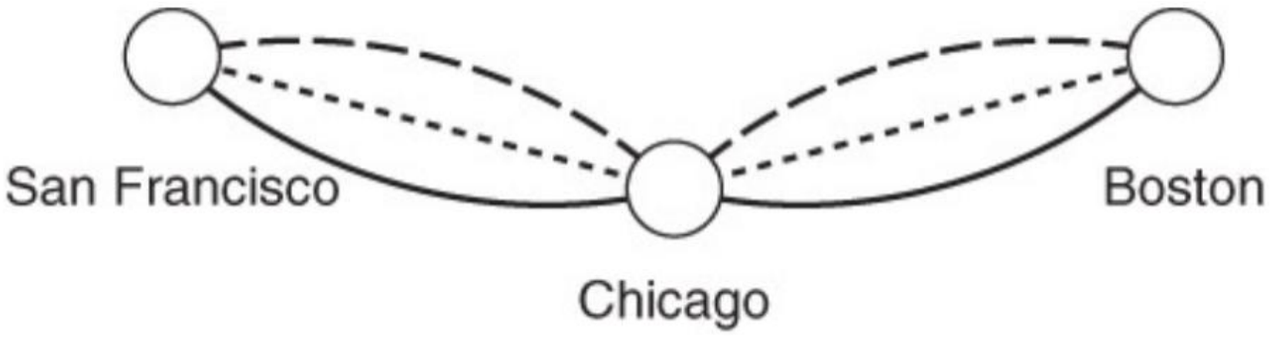
A bipartite graph, or bigraph, is a graph with two distinct sets of vertices where each vertex in one set is only connected to vertices in the other set.



Bipartite graphs are useful when modeling relationships between different types of objects. For example, one set of vertices might represent businesses and another might represent people. An edge between a given person and a business appears if the person works for that business. Other examples include teachers and students, members and groups, and train cars and trains.

MULTIGRAPH

A multigraph is a graph with multiple edges between vertices. Let's take a shipping company as an example.



The company could use a graph database for determining the least costly way to ship items between cities. Multiple edges between cities could represent various shipping options, such as shipping by truck, train, or plane. Each edge would have its own properties, such as the time taken to transport an item between two cities, cost per kilogram to ship, and so forth.

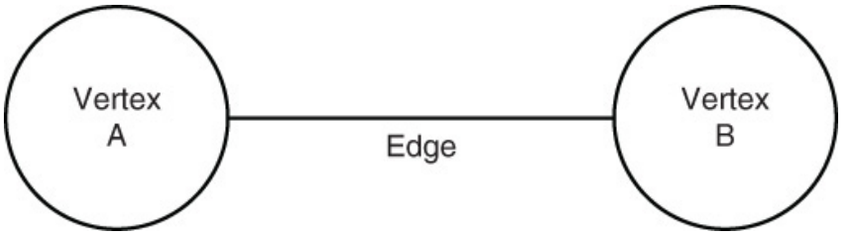
DEFINITION

A graph database is based on a branch of mathematics known as graph theory. The techniques in this area of mathematics are useful for analyzing connections and links between entities. As you shall see, these techniques are quite useful in many data management areas.

Graphs are mathematical objects that consist of two parts: **vertices and edges**.

Vertices represent things. They could be just about anything, including

- Cities
- Employees in a company
- Proteins
- Electrical circuits
- Junctions in a water line
- Organisms in an ecosystem
- Train stations



Vertex: specifies an entity. Usually, entities in a graph belongs to the same category.

Edge: specifies a relation between two vertices. Relations may be long terms or short terms.

Cities are connected to other cities by roads. Employees work with other employees. Proteins interact with other proteins. Electrical circuits are linked to other electrical circuits. Junctions in water lines connect to other junctions. Organisms in ecosystems are predators and prey of other organisms. Train stations are connected to other train stations by railway lines.

The links or connections between entities are represented by edges. This might seem like an obvious representation for some relations, such as roads and railway lines between cities. However, it might be less

obvious in other cases, such as interacting proteins and organisms in an ecosystem. The flexible nature of vertices and edges makes them well suited to model both concrete and abstract relations between things.

Vertices and edges may have **properties**.

Some example:

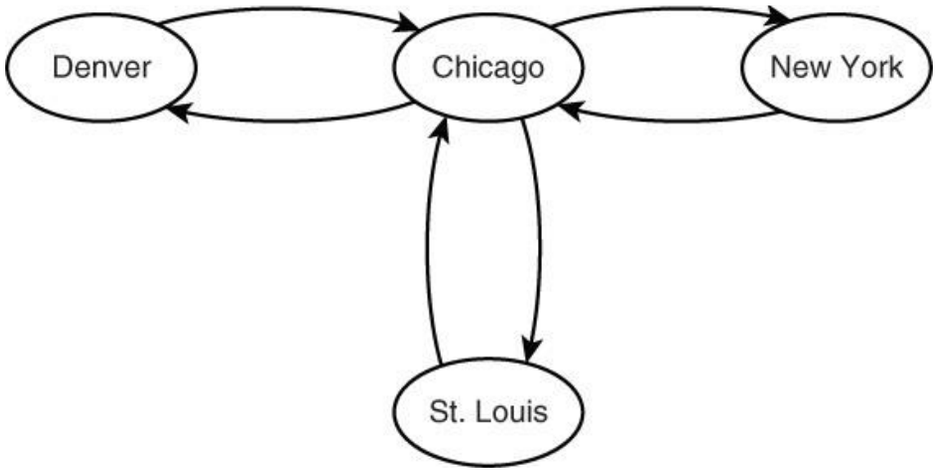
MODELING GEOGRAPHIC LOCATIONS

Highways and railways have two distinct properties of interest here: They are designed to link geographic locations and they last a long time.

Geographic locations are modeled as vertices. These could be cities, towns, or intersections of highways. Vertices have properties, like names, latitudes, and longitudes. In the case of towns and cities, they have populations and size measured in square miles or kilometers.

Highways and railways are modeled as edges between two vertices. They also have properties, such as length, year built, and maximum speed. Highways could be modeled in two ways. A highway could be a single edge between two cities, in which case it models the road traffic in both directions. Alternatively, a graphical representation could use two edges, one to represent travel in each direction (for example, east to west and west to east). Which is the “right way” to model highways? It depends.

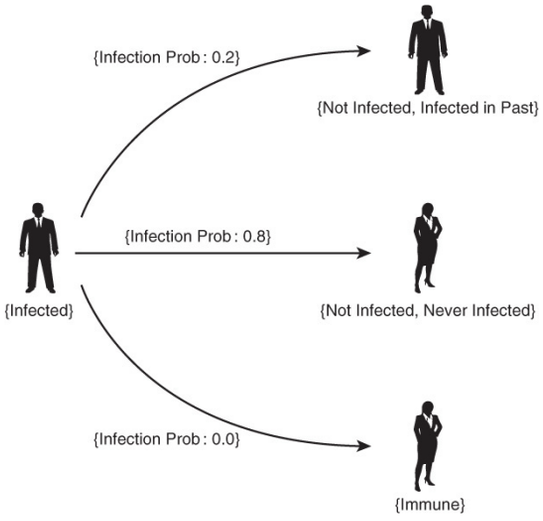
If your goal is to model distance and approximate travel times between cities, then a single edge might be sufficient. If you are interested in more detailed descriptions of highways, such as direction, number of lanes, current construction areas, and locations of accidents, then using two edges is a better option. When you use two edges between cities, it helps to indicate which direction traffic is flowing. This is done with a type of edge known as a directed edge.



MODELING INFECTIOUS DISEASES

Infectious diseases can spread from person to person. For example, a person coughs into his hand and bacteria and viruses are left on his hand. When that person shakes the hand of someone else, there is a chance that a pathogen is transmitted to the other person, who may eventually become infected. The spread of infectious disease is readily modeled using graphs.

Vertices represent people, whereas edges represent interactions between people, such as shaking hands or standing in close proximity. Both the vertices and the edges have properties that help represent the way diseases spread.



People have properties, such as age and weight. In the case of the infectious disease model, the most important property is infection status, which could be

- Not infected now, never infected before
- Not infected now, infected in the past
- Infected now
- Immune

You want to keep track of these properties because they influence the probability of becoming infected:

- If you are not infected now and never have been, you have a moderately high probability of becoming infected upon contact with an infected person.
- If you are not infected now but were infected in the past, you have probably acquired some immunity to the infectious disease. This means you have a low probability of getting infected upon contact with an infected person.
- If you are infected now and come in contact with another infected person, you will both continue to be infected. There is no change.
- If you are immune, either because of some natural immunity or medical immunization, then you will not become infected upon contact with another infected person.

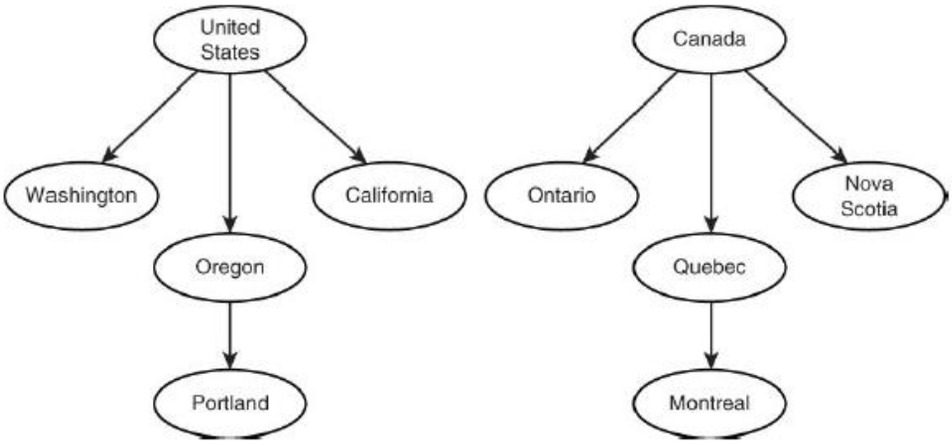
The state of an infectious disease graph would change frequently as people interact; some people become infected while others recover from the disease. As you can see, this is much different from the railways and highways example, which is fairly static in terms of nodes and edges. Properties of cities and roads may change as populations change and car accidents occur. The infectious disease graph changes as people interact, something that happens frequently and rapidly.

Edges—or in this case, interactions between people—have properties. For example, there is a probability that someone will transmit a disease to another person by shaking hands. This interaction has a higher rate of transmission than two people standing in close proximity but not touching. Some pathogens require physical contact to spread disease, whereas other airborne diseases can transmit without direct contact. These are the kinds of properties that would be associated with edges.

Graphs are useful for more than modeling railways and disease transmission. Sometimes there is no flow or transmission of objects between vertices. Instead, some graphs model relations between things that persist over time.

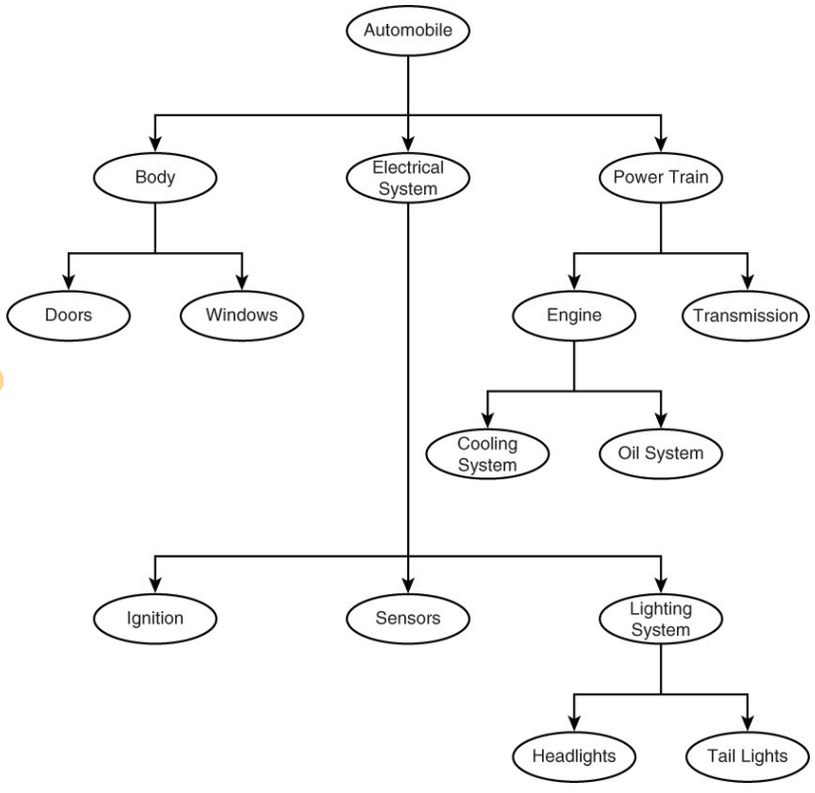
MODELING ABSTRACT AND CONCRETE ENTITIES

Graphs are well suited to model abstract relations, like a part-of relation. For example, the state of Oregon is a part of the United States, and the province of Quebec is a part of Canada. The city of Portland is located in Oregon, and the city of Montreal is located in Quebec. This kind of hierarchical relationship is modeled in a special type of graph known as a tree.



A tree has a special vertex call the root. The root is the top of the hierarchy. These two trees, one for the United States and one for Canada. Both show the relationship between national, regional, and local government entities.

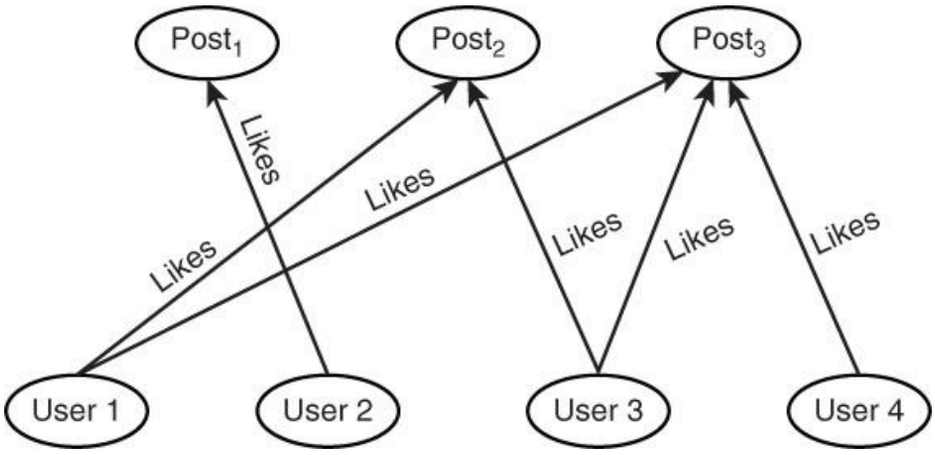
Notice that **all nodes connect up to only one other vertex**. The upper vertex is often called the parent vertex, and the lower vertices are called children vertices. Parent vertices can have multiple children vertices. Trees are useful for modeling hierarchical relationships, such as organization charts, as well as part-of relations, such as parts of a car.



MODELING SOCIAL MEDIA

Social networking sites like Facebook and LinkedIn allow users to interact and communicate with each other online. They have extended the way people communicate by introducing new ways of interacting, such as the “Like” button. This makes it quick and easy to indicate you like or appreciate someone else’s post.

A social media "like" can be modeled as a link between a person and a post. Many people can like the same post, and people can have multiple posts each with a different number of likes. The vertices in this case would be people and posts. It is worth pointing out, not all vertices and edges have to be of the same type; there can be a mix of different types in a single graph.



The figure shows an example of a people-like-posts graph. You will notice, unlike many other graphs, this has a special property. The edges only go from people to posts; there are no edges between people or between posts. This is a special type of graph known as a bipartite graph, and it is useful for modeling relations between different types of entities.

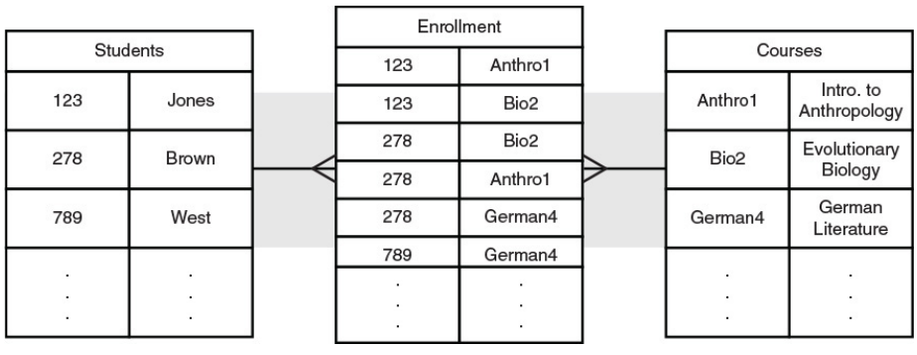
ADVANTAGES OF GRAPH DATABASES

Graph databases show explicit relations between entities. Vertices represent entities, and they are linked or connected by edges. In relational databases, connections are not represented as links. Instead, two entities share a common attribute value, which is known as a key.

- Graph databases adopt vertices and edges to **store** explicit **relations** between **entities**.
- In relational databases, connections are not represented as links. Instead, two entities share a common attribute value, which is known as a key.
- Join operations are used in relational databases to find connections or links.
- When dealing with huge amount of data (several tables with too much rows) join operation became computationally expensive.

QUERY FASTER BY AVOIDING JOINS

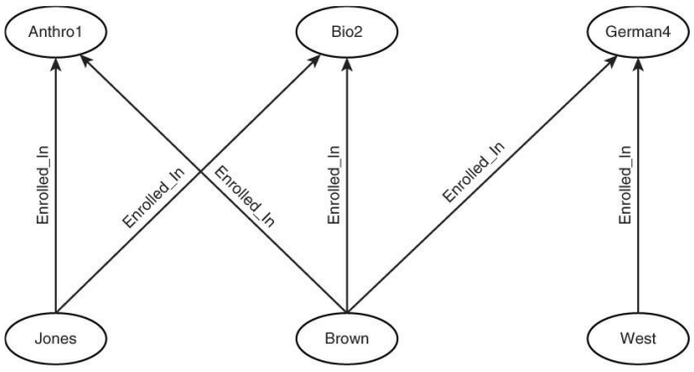
To find connections or links in a relational database, you must perform an operation called a join. A join entails looking up a value from one table in another table. The table Students has a list of student names and IDs. The student ID is also used in the table Enrollment to indicate a student is enrolled in a course. To list all of the courses a student is enrolled in, you need to perform a join between two tables. This can be time consuming when using join operations frequently on large tables.



Students	
123	Jones
278	Brown
789	West
.	.
.	.
.	.

Enrollment	
123	Anthro1
123	Bio2
278	Bio2
278	Anthro1
278	German4
789	German4
.	.
.	.
.	.

Courses	
Anthro1	Intro. to Anthropology
Bio2	Evolutionary Biology
German4	German Literature
.	.
.	.
.	.



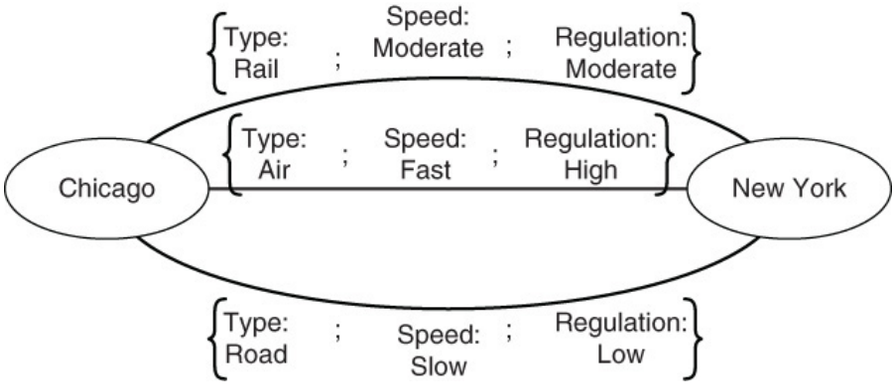
Alternatively, you can represent relations between a student and a course using a graph, as shown in Figure. The edges between students and courses allow users to quickly query all the courses a particular student is enrolled in.

In a graph database, instead of performing joins, you follow edges from vertex to vertex. This is a much simpler and faster operation. Graph databases allows to modeling many-to-many relations in a easier way than relational databases. Edges allow us to explicitly modeling many-to-many relations, rather than using tables.

Graphs have even made their way into popular entertainment. Take the board game Pandemic from Z-Man Games, for instance. In Pandemic, players work together to suppress and eradicate four different infectious diseases, represented by colored plastic blocks. The game board displays numerous major cities across the world, and each city possesses between two and six lines connecting it to another city. Each line represents a pathway for transmission of a given disease. Throughout the game, players utilize unique powers assigned to them at the beginning of the game to remove the colored blocks from the board.
As you can see, Pandemic's gameplay takes place within a graphical model. Cities act as the vertices, whereas the lines serve as edges. You may also attribute certain properties to the vertices and edges. For instance, each city has a certain number of lines connecting it to other cities, as well as a level of infection, represented by the number of colored blocks on a city. In this case, properties of the edges are more abstract. They include whether or not the lines connect two infected cites, two healthy cities, or one infected city and one healthy city.

MULTIPLE RELATIONS BETWEEN ENTITIES

Using multiple types of edges allows database designers to readily model multiple relations between entities. This is particularly useful when modeling transportation options between entities. For example, a transportation company might want to consider road, rail, and air transportation between cities. Each has different properties, such as time to deliver, cost, and government regulations.



Multiple relations can be modeled in relational databases, but they are explicit and easy to understand when using a graph database.

DESIGN GRAPH DATABASE

In general, data base designer should have in mind the main kinds of queries that the application will perform on the data.

Graph databases are well suited to problem domains that are easily described in terms of entities and relations between those entities. Entities can be virtually anything, from proteins to planets. Of course, other NoSQL databases and relational databases are well suited to modeling entities, too.

Graph database applications frequently include queries and analysis that involve

- Identifying relations between two entities
- Identifying common properties of edges from a node
- Calculating aggregate properties of edges from a node
- Calculating aggregate values of properties of nodes

Typical Queries for Graph Databases (examples of each of these types of queries):

- How many hops (that is, edges) does it take to get from vertex A to vertex B?
- How many edges between vertex A and vertex B have a cost that is less than 100?
- How many edges are linked to vertex A?
- What is the centrality measure of vertex B?
- Is vertex C a bottleneck; that is, if vertex C is removed, which parts of the graph become disconnected?

These queries are different from queries associated with document and column family databases. There is **less emphasis on selecting** by particular properties, for example, how many vertices have at least 10 edges?

Similarly, there is **less emphasis on aggregating** values across a group of entities. For example, in a column family database, you might have a query that selects all customer orders from the Northeast placed in the last month and sum the total value of those orders. These types of queries can be done in graph databases, but they do not reflect the flexibility and new ways of querying offered by graph databases.

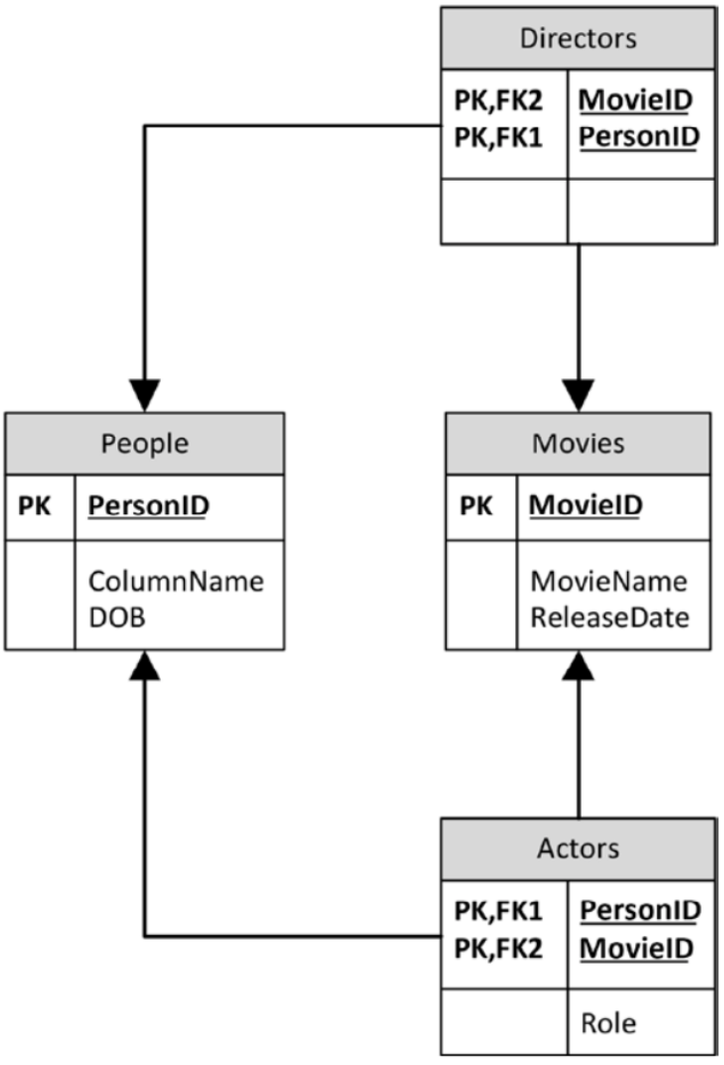
The queries listed above are fairly abstract. They are stated in the terminology used by computer scientists and mathematicians when working with graphs. Designer needs to translate queries from the problem domain to the domain of experts working on graphs.

AN EXAMPLE – HOLLYWOOD-DB

Let suppose to design a DB for handling movies, actors and directors.

Let suppose to offer the following services for users:

1. Find all the actors that have ever appeared as a co-star in a movie starring a specific actor (for example: Keanu Reeves).
2. Find all movies and actors directed by a specific director (for example Andy Wachowski).
3. Find all movies in which specific actor has starred in, together with all co-stars and directors



While the relational model can easily represent the data that is contained in a graph model, we face two significant problems in practice:

1) SQL lacks the syntax to easily perform graph traversal, especially traversals where the depth is unknown or unbounded. For instance, using SQL to determine friends of your friends is easy enough, but it is hard to solve the “degrees of separation” problem (often illustrated using the example of the number of connections that separate you from Kevin Bacon).
2) Performance degrades quickly as we traverse the graph. Each level of traversal adds significantly response to a specific query.

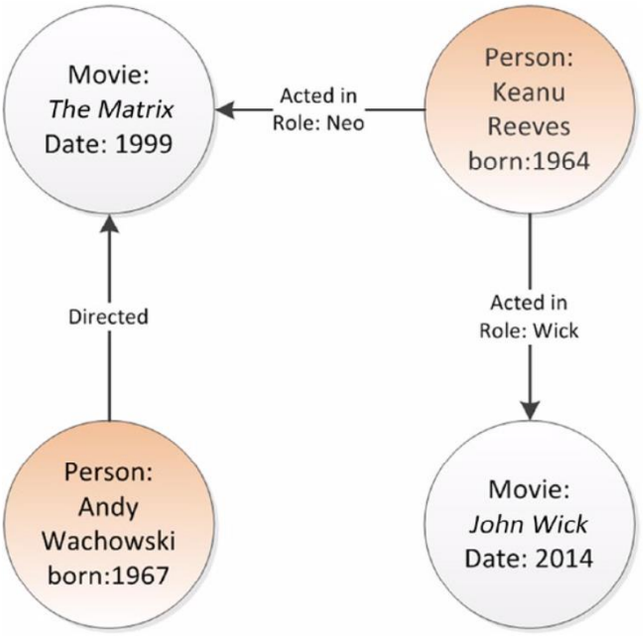
SQL query for finding all actors who have ever worked with Keanu Reeve

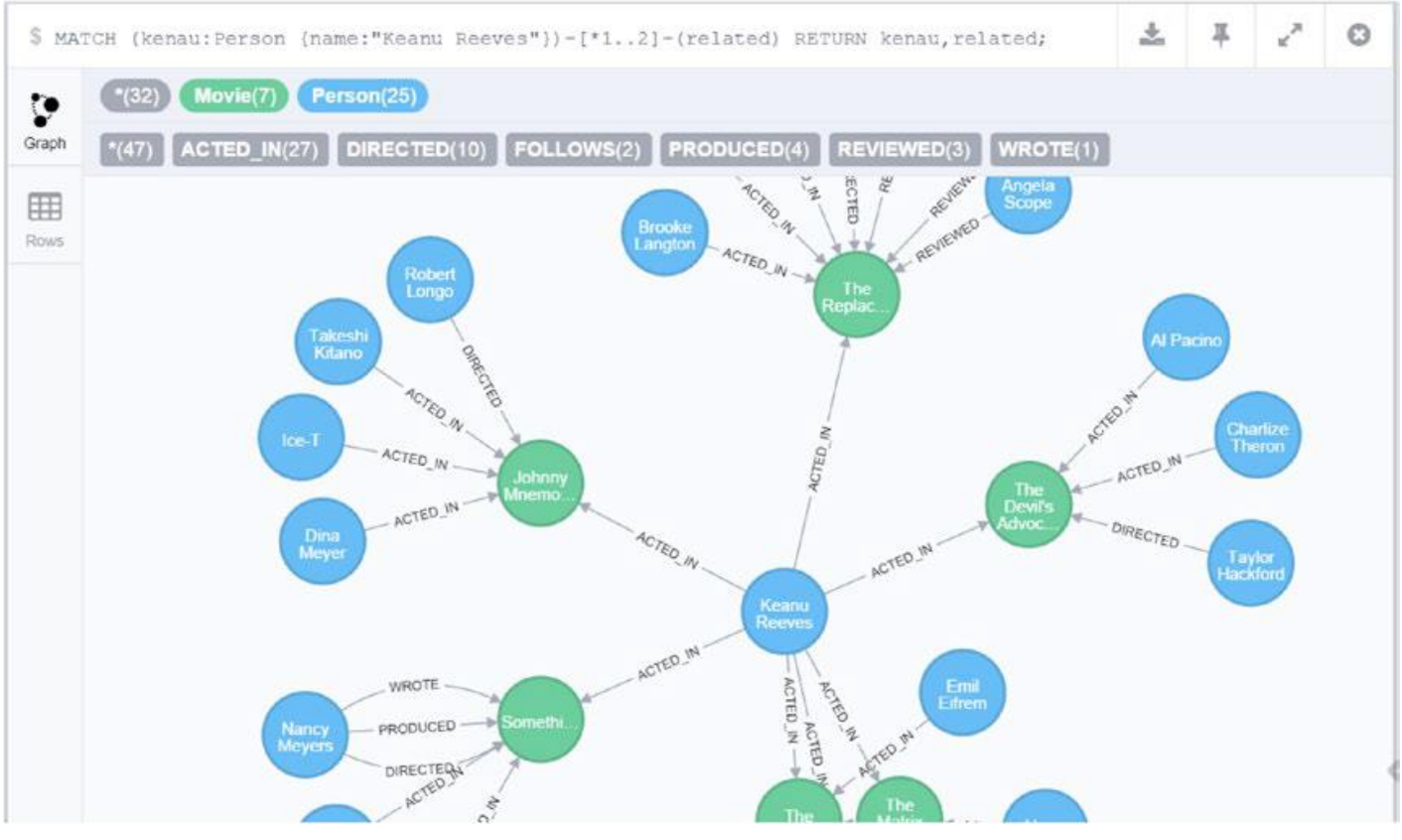
```
SELECT p2.personname, m1.movieName
  FROM people p1
       JOIN actors a1 ON (p1.personid = a1.personid)
       JOIN movies m1 ON (a1.movieid = m1.movieid)
       JOIN actors a2 ON (a2.movieid = m1.movieid)
       JOIN people p2 ON (p2.personid = a2.personid)
 WHERE p1.personname = 'Keanu Reeves';
```

Performance in an RDBMS is also a potential issue. Providing that appropriate indexes are created, our sample query above will execute with acceptable overhead. But each join requires an index lookup for each

actor and movie. Each index lookup adds overhead and performance that for deep graph traversals are often unacceptable. Sometimes the best solution is to load the tables in their entirety into map structures within application code and traverse the graph in memory. However, this assumes that there is enough memory in application code to cache all the data, which is not always the case.

Because key-value stores do not support joins, they offer even less graph traversal capability than the relational database. In a pure key-value store or document database, graph traversal logic must be implemented entirely in application code.





DESIGNING A SOCIAL NETWORK GRAPH DATABASE

Use Cases: Let consider the database developer as the main actor of the social network. The main use cases may be:

1. Join and leave the site
2. Follow the postings of other developers
3. Post questions for others with expertise in a particular area
4. Suggest new connections with other developers based on shared interests
5. Rank members according to their number of connections, posts, and answers

The model will start simple with just two entities: developers and posts. You can always add more later, but it helps to flesh out the relations and properties of a small number of entities at a time.

Properties of **developers** include:	Properties of **posts** include:
• Name • Location • NoSQL databases used • Years of experience with NoSQL databases • Areas of interest, such as data modeling, performance tuning, and security	• Date created • Topic keywords • Post type (for example, question, tip, news) • Title • Body of post

Next, consider the relations between entities. Entities may have one or more relations to other entities. Because there are two types of entities, there can be four possible combinations of entity-relation-entity:

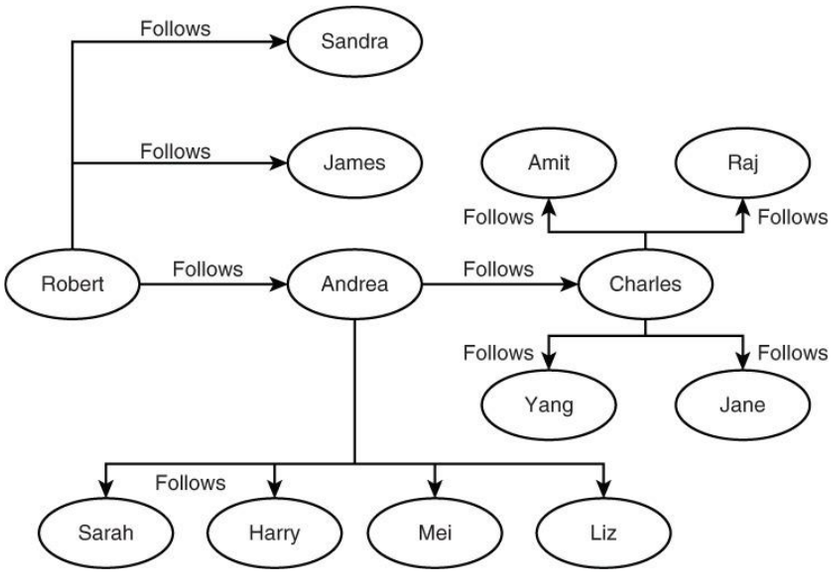
- Developer-relation-developer
- Developer-relation-post
- Post-relation-developer
- Post-relation-post

The term relation is a placeholder in the preceding list. As a graph database designer, one of your first tasks is to identify each of these relations.

It helps to consider all possible combinations of entities having relations when the number of entity types is small. Some of the combinations may not be relevant to the types of queries you will pose. You can eliminate them from consideration. This process helps reduce the chance that you miss a relevant relation early in the design phase.

As the number of entities grows, you might want to focus on combinations that are reasonably likely to support your queries.

- **Developer – follows – Developer**



If a developer follows another developer, it is reasonable that he/she is interested in reading his/her posts.

The depth of the path, namely how many edges to traverse for identifying a followed developer, is a parameter that can be defined and handled at the level of application code.

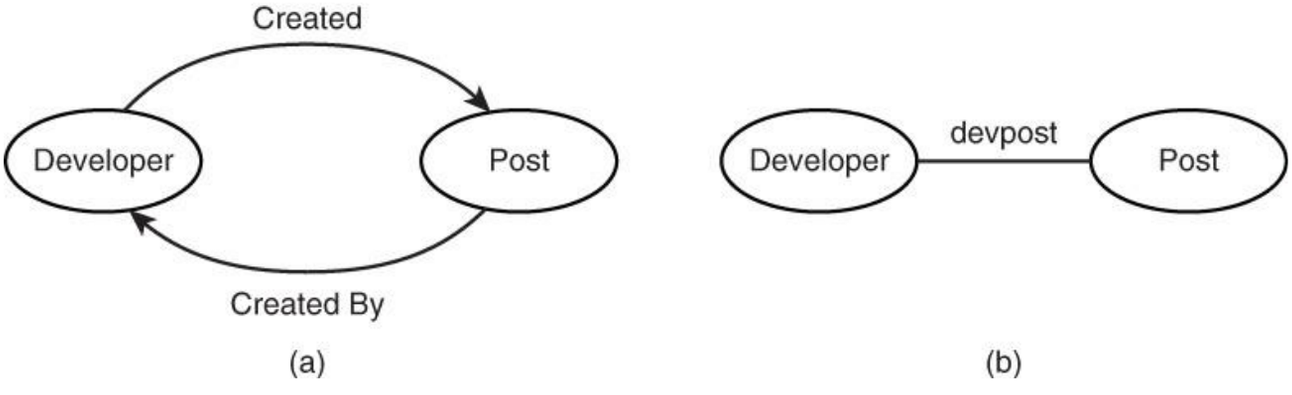
If Robert Smith follows Andrea Wilson, then Robert will see all of Andrea's posts when he logs on to the NoSQL social network. The site designers believe the followers of a developer might be interested in who that developer follows as well. For example, if Andrea follows Charles Vita, then Robert should see Charles's posts as well.

You can imagine adding posts from the developers Charles follows, but that could start to overwhelm Robert's feed.

- **Relations between Developer and Post**

The relation between developers and posts is "created"; that is, developers create posts. This implies a reverse relationship; that is, that posts are "created by" developers. There are two ways to model this:

- A designer could create a directed edge of type "created" from the developer vertex to the post vertex and another directed edge from the post to the developer of type "created by,".
- Because one of these relations always implies the other, you can avoid using two directed edges by using a single, possibly undirected, edge.



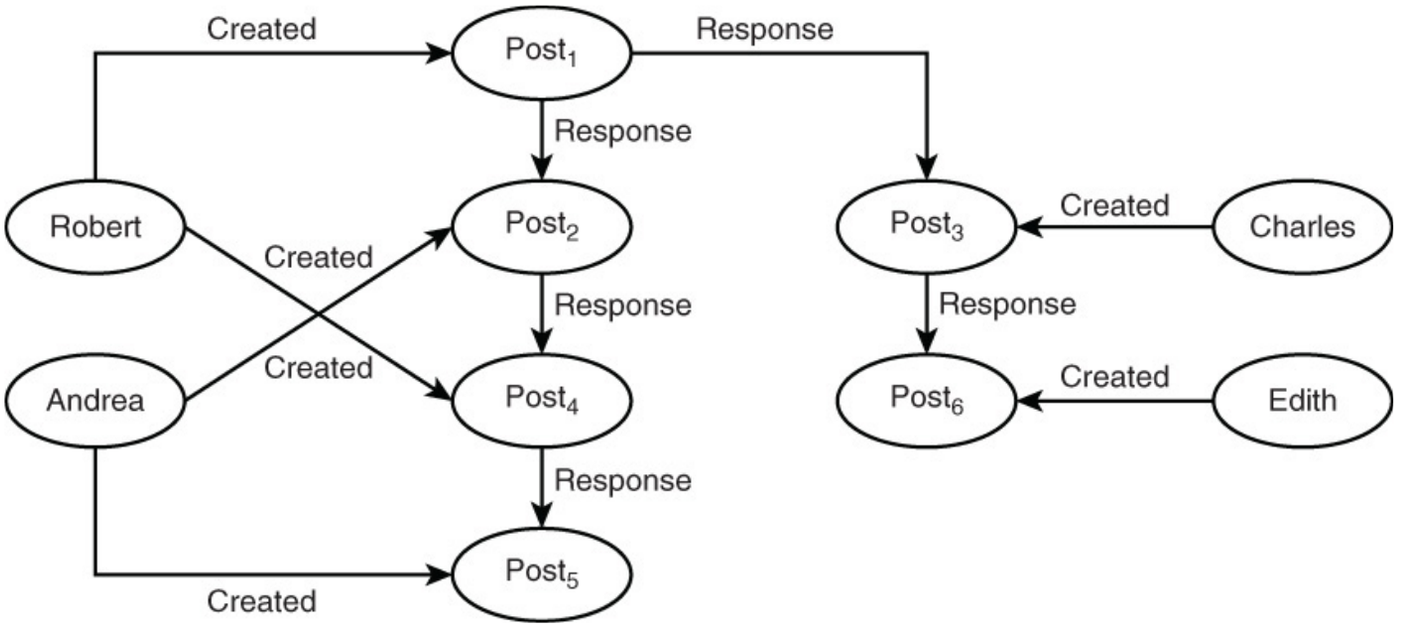
- **Post-Post Relation**

There is no one correct way to model a graph database for all possible problems. If you have queries that frequently involved gathering posts and then looking up the creators of those posts, it makes sense to have a created by relation.

By creating such an edge, you implement a direct link between the post and the developer. It is this feature that allows graph database designers to avoid working with joins to retrieve data from related entities. Following edges between vertices is a simple and fast operation, so it is possible to follow long paths or a large number of paths between vertices without adversely impacting performance.

At first glance, the post-relation-post may not appear useful. After all, posts do not create other posts. However, there is no rule that says all relations between entities need to be the same type. In fact, one of the powerful features of graph database modeling is the ability to use different types of relations. For example, a post may be created in response to another post. This is particularly useful for questions.

Imagine that Robert posted the question: Is there a faster way than Dijkstra's algorithm to find the shortest path? Andrea and Charles might each reply with their own answers. Robert then posts another question to clarify his understanding of Andrea's response. Andrea responds with additional details. Meanwhile, Edith Woolfe adds additional details to Charles's post. The resulting graph of posts is a tree with Robert's initial post as the root and branches that follow the two parts of the conversation thread.



MAPPING QUERIES

Abstract queries map to useful queries about graph databases. Some queries are based on paths, such as the distance between two nodes; for example, "How many follows relations are between Developer A and Developer B?" Other queries take into account the global structure of the graph, such as "If a developer left the social network, would there be disconnected groups of developers?"

Graph-Centric Query	Domain-Specific Query
How many hops (that is, edges) does it take to get from vertex A to vertex B?	How many follows relations are between Developer A and Developer B?
How many incoming edges are incident to vertex A?	How many developers follow Andrea Wilson?
What is the centrality measure of vertex B?	How well connected is Edith Woolfe in the social network?
Is vertex C a bottleneck; that is, if vertex C is removed, which parts of the graph become disconnected?	If a developer left the social network, would there be disconnected groups of developers?

When you design graph databases, you start with domain-specific queries. Ultimately, you will want to map these domain-specific queries to graph-specific queries that reference vertices, edges, and graph measures, like centrality and betweenness.

When you have your domain-specific queries mapped to graph-specific queries, you have the full range of graph query tools and graph algorithms available to you to analyze and explore your data.

BASIC STEPS FOR DESIGNING GRAPHDB

The following are the basic steps to getting started with graph database design:

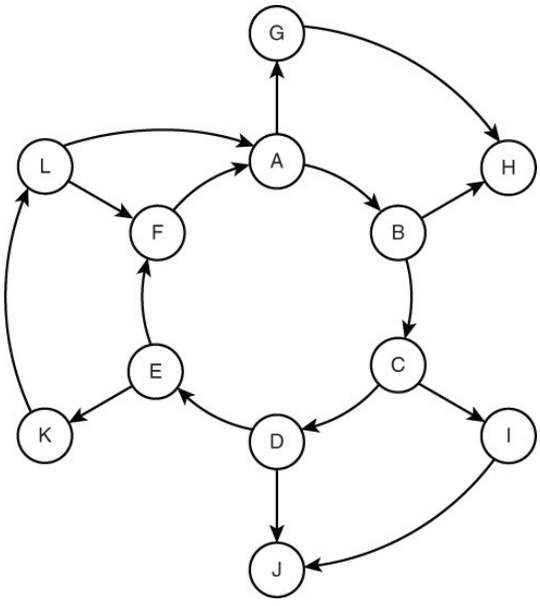
- Identify the queries you would like to perform.
- Identify entities in the graph.
- Identify relations between entities
- Map domain-specific queries to more abstract queries so you can implement graph queries and use graph algorithms to compute additional properties of nodes.

SOME ADVICES

1. Pay attention to the dimension of the graph: the bigger the graph the more computational expensive will be performing queries.
2. If available, define and use indexes for improving retrieval time.
3. Use appropriate type of edges:
    a. In case of symmetrical relations, use undirected edge, otherwise use direct edges.
    b. Properties requires memory! If possible, use simple edge without properties. Pay also attention to the data type of the properties.

WATCH FOR CYCLES WHEN TRAVERSING GRAPHS

In order to avoid endless cycles, use appropriate data structure for keeping track of visited nodes.



Cycles are paths that lead back to themselves. The graph has a cycle A-B-C-D-E-F-A.

If you were to start a traversal at vertex A and then followed a path through vertices B, C, D, E, and F, you would end up back at A. If there is no indication that you have already visited A, you could find yourself in an endless cycle of revisiting the same six nodes over and over again.

Not all graphs have them—trees, for example, do not.

If you might encounter cycles, keep track of which vertices have already been visited. This can be as simple as maintaining a set named visitedNodes.

Each time you visit a node, you first check to see if that node is in the set visitedNodes. If it is, you return with processing the node; otherwise, you process the node and add it to the set.

SCALABILITY GRAPH DATABASE

The graph database systems available today can work with millions of vertices and edges using a single server. You should consider how your applications and analysis tools will scale as the following occurs (the developer must take a special attention to the growing of):

- The number of nodes and edges grow
- The number of users grow
- The number and size of properties on vertices and edges grow

Increases in each of these three areas can put additional demands on a database server. Most of the famous implementation of graph databases do not consider the deployment on cluster of servers (**no data partitions nor replicas**). Indeed, they are designed for running on a single server and can be only vertically scaled.

EXECUTION TIMES OF TWO TYPICAL ALGORITHMS

Also consider the algorithms you use to analyze data in the graph. The time required to perform some types of analysis grows rapidly as you add more vertices. For example, Dijkstra's algorithm for finding the shortest paths in a network takes a time related to the square of the number of vertices in a graph.

To find the largest group of people who all follow each other (known as a maximal clique) in the NoSQL social network requires time that grows exponentially with the number of people.

The table shows examples of time required to run Dijkstra's algorithm and solve the maximal clique problem. Be careful when using graph algorithms. Some that run in reasonable amounts of time with small graphs will not finish in reasonable amounts of time if the graphs grow too large. This table assumes a time of two units to complete the operations on a single vertex.

Number of Vertices	Time to Find Shortest Path	Time to Find Maximal Clique
1	2	2
10	200	1,024
20	800	1,048,576
30	1,800	1,073,741,824
40	3,200	1,099,511,627,776
50	5,000	1,125,899,906,842,620

IN MEMORY DATABASE

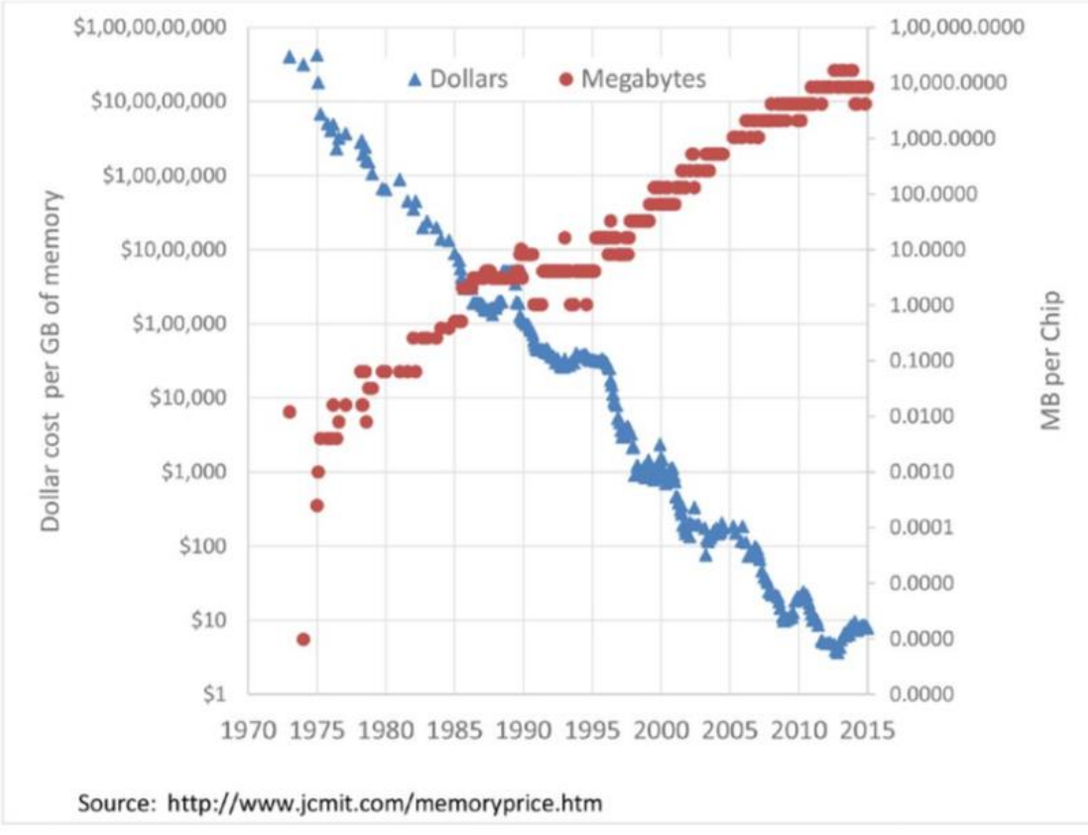
Ever since the birth of the first database systems, database professionals have strived to avoid disk IO at all costs. IO to magnetic disk devices has always been many orders of magnitude slower than memory or CPU access, and the situation has only grown worse as Moore's law accelerated the performance of CPU and memory while leaving mechanical disk performance behind.

The emergence of affordable solid state disk (SSD) technology has allowed for a quantum leap in database disk performance. Over the past few years, SSD technology has shifted from an expensive luxury to a mainstream technology that has a place in almost every performance-critical database system. However, SSDs have some unique performance characteristics and some database systems have been developed to exploit these capabilities.

While SSDs allow IO to be accelerated, the increasing capacity and economy of server memory sometimes allows us to avoid IO altogether. Many smaller databases can now fit entirely within the memory capacity of a single server, and certainly within the memory capacity of a cluster. For these databases, an in-memory solution may be even more attractive than an SSD architecture.

The solid state disk may have had a transformative impact on database performance, but it has resulted in only incremental changes for most database architectures. A more paradigm-shifting trend has been the increasing practicality of storing complete databases in main memory.

The cost of memory and the amount of memory that can be stored on a server have both been moving exponentially since the earliest days of computing. Figure 7-4 illustrates these trends: both the cost of memory per unit storage and the amount of storage that can fit on a single memory chip have been increasing over many decades (note the logarithmic scale—the relatively straight lines indicate exponential trends).



NOT ONLY BIG DATA AND SCALABLE SOLUTIONS

The design of a database solution is driven by the application requirements.

Thus, if our application does not need to handle huge amount of data and to scale, both vertically and horizontally, we can think about adopting different solutions that the ones discussed so far.

In-memory databases may be suitable for applications whose handled data may fit all in the memory of a single server.

Moreover, there also can be databases that can reside in the memory capacity of a cluster (if replication is needed).

The size of the average database—particularly in light of the Big Data phenomenon—has been growing exponentially as well. For many systems, database growth continues to outpace memory growth. But many other databases of more modest size can now comfortably be stored within the memory of a single server.

And many more databases than that can reside within the memory capacity of a cluster. Traditional relational databases use memory to cache data stored on disk, and they generally show significant performance improvements as the amount of memory increases. But there are some database operations that simply must write to a persistent medium. In the traditional database architecture, COMMIT operations require a write to a transaction log on a persistent medium, and periodically the database writes “checkpoint” blocks in memory to disk. Taking full advantage of a large memory system requires an architecture that is aware the database is completely memory resident and that allows for the advantages of high-speed access without losing data in the event of a power failure. There are two changes to traditional database architecture an in-memory system should address:

- **Cache-less architecture**: Traditional disk-based databases almost invariably cache data in main memory to minimize disk IO. This is futile and counterproductive in an in-memory system: there is no point caching in memory what is already stored in memory!
- **Alternative persistence model**: Data in memory disappears when the power is turned off, so the database must apply some alternative mechanism for ensuring that no data loss occurs.

FEATURES OF IN-MEMORY DATABASES

Very **fast** access to the data, avoiding the bottleneck of the I/O transfers.

Suitable for application that do not need to write continuously to a persistent medium.

Ad-hoc architecture for being aware that its data are resident in memory and for avoiding the data loss in the event of a power failure (alternative persistency model).

Cache-less architecture: despite traditional disk-based databases, a memory caching systems is not need (every thing is already in memory!!!).

SOLUTIONS FOR DATA PERSISTENCY

In-memory databases generally use some combination of techniques to ensure they don't lose data. These include:

- **Replicating data** to other members of a cluster.
- Writing complete database **images** (called snapshots or checkpoints) to disk files.
    - frequentily, using event based approach, copy the memory to the disk
- Writing out transaction/operation records to an append-only disk file (called a **transaction log** or **journal**).
    - usato per memorizzare le informazioni, simile al commit log per i columnar database.

TIMESTEN (ORACLE SOLUTION)

TimesTen implements a fairly familiar **SQL-based** relational model.

In a TimesTen database, all data is **memory resident**. Persistence is achieved by **writing periodic snapshots** of memory to disk, as well as writing to a disk-based transaction log following a transaction commit.

In the default configuration, all disk writes are asynchronous: a database operation would normally not need to wait on a disk IO operation. However, if the power fails between the transaction commit and the time the transaction log is written, then data could be lost. This behavior is not ACID compliant because transaction durability (the "D" in ACID) is not guaranteed. However, the user may choose to configure synchronous writes to the transaction log during commit operations. In this case, the database becomes ACID compliant, but some database operations will wait on disk IO.

The TimesTen architecture

1) When the database is started, all data is loaded from checkpoint files into main memory (snapshot)
2) The application interacts with TimesTen via SQL requests that are guaranteed to find all relevant data inside that main memory
3) Periodically or when required database data is written to checkpoint files
4) An application commit triggers a write to the transaction log, though by default this **write** will be **asynchronous** so that the application will not need to wait on disk [contiene le informazioni sulle transazioni in modo da recuperarle in caso di fallimento.].
5) The transaction log can be used to recover the database in the event of failure.

REDIS

While TimesTen is an attempt to build an RDBMS compatible in-memory database, Redis is at the opposite extreme: essentially an in-memory **key-value store**. Redis (Remote Dictionary Server) was originally envisaged as a simple in-memory system capable of sustaining very high transaction rates on underpowered systems, such as virtual machine images.

Redis was created by Salvatore Sanfilippo in 2009.

Redis follows a familiar key-value store architecture in which keys point to objects. In Redis, objects consist mainly of strings and various types of collections of strings (lists, sorted lists, hash maps, etc.). Only primary key lookups are supported; Redis does not have a secondary indexing mechanism.

Although Redis was designed to hold all data in memory, it is possible for Redis to operate on datasets larger than available memory by using its **virtual memory feature**. When this is enabled, Redis will "swap out" older key values to a disk file. Should the keys be needed they will be brought back into memory. This option obviously involves a significant performance overhead, since some key lookups will result in disk IO.

Redis uses disk files for persistence:

- The Snapshot files store copies of the entire Redis system at a point in time. Snapshots can be created on demand or can be configured to occur at scheduled intervals or after a threshold of writes has been reached. A snapshot also occurs when the server is shut down.
- The Append Only File (AOF) keeps a journal of changes that can be used to "roll forward" the database from a snapshot in the event of a failure. Configuration options allow the user to configure writes to the AOF after every operation, at one-second intervals, or based on operating-system-determined flush intervals.

In addition, Redis supports asynchronous **master/slave replication**. If performance is very critical and some data loss is acceptable, then a replica can be used as a backup database and the master configured with minimal disk-based persistence. However, there is no way to limit the amount of possible data loss; during high loads, the slave may fall significantly behind the master.

The architectural components:

1) The application interacts with Redis through primary key lookups that return "values"—strings, sets of strings, hashes of strings, and so on
2) The key values will almost always be in memory, though it is possible to configure Redis with a virtual memory system, in which case key values may have to be swapped in or out
3) Periodically, Redis may dump a copy of the entire memory space to disk
4) Additionally, Redis can be configured to write changes to an append-only journal file either at short intervals or after every operation
5) Finally, Redis may replicate the state of the master database asynchronously to slave Redis servers

Although Redis was designed from the ground up as in an in-memory database system, applications may have to wait for IO to complete under the following circumstances:

- If the Append Only File is configured to be written after every operation, then the application needs to wait for an IO to complete before a modification will return control.
- If Redis virtual memory is configured, then the application may need to wait for a key to be "swapped in" to memory.

Redis is popular among developers as a simple, high-performance key-value store that performs well without expensive hardware. It lacks the sophistication of some other nonrelational systems such as MongoDB, but it works well on systems where the data will fit into main memory or as a caching layer in front of a disk-based database.

HANA DB (A SAP PRODUCT) Da leggere tutto

SAP introduced HANA in 2010, positioning it as a revolutionary in-memory database designed primary for Business Intelligence (BI), but also capable of supporting OLTP workloads.

SAP HANA is a **relational database** designed to provide breakthrough performance by combining in-memory technology with a columnar storage option, installed on an optimized hardware configuration.

Tables in HANA can be configured for row-oriented or columnar storage. Typically, tables intended for BI purposes would be configured as columnar, while OLTP tables are configured as row oriented. The choice of row or columnar formats provides HANA with its ability to provide support for both OLTP and analytic workloads.

Data in the row store is guaranteed to be in memory, while data in the column store is by default loaded on demand. However, specified columns or entire tables may be configured for immediate loading on database startup.

[Classical row tables are exploited for OLTP operations and are stored in memory.

Columnar tables are often used for OLAP operations. By default, they are loaded in memory by demand. However, the database may be configured to load in memory a set of columns or entire tables since the beginning.]

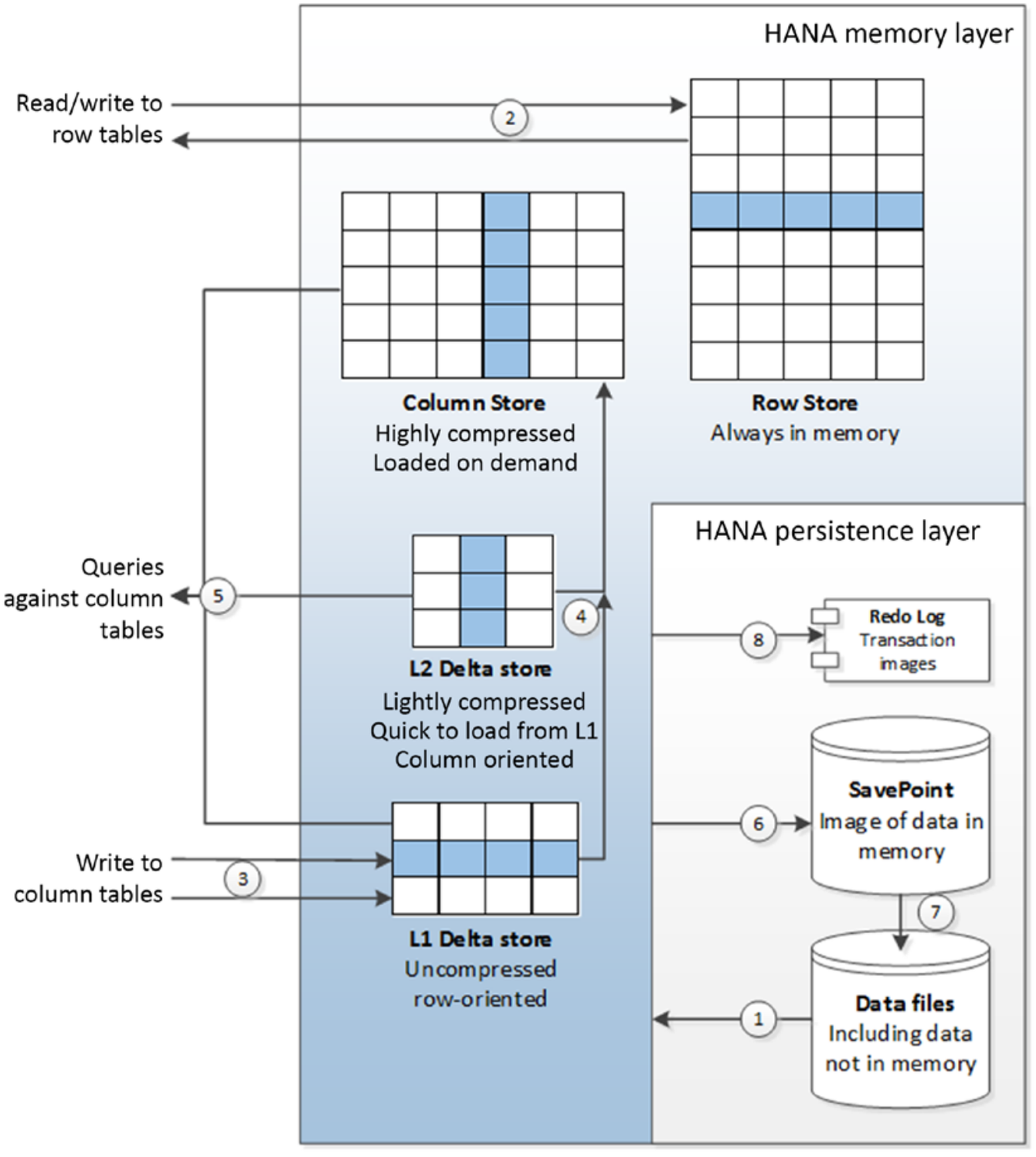
The persistence architecture of HANA uses the snapshot and journal file pattern found in Redis and TimesTen. HANA periodically snapshots the state of memory to Savepoint files. These Savepoints are periodically applied to the master database files.

ACID transactional consistency is enabled by the transaction "redo" log. As with most ACID-compliant relational databases, this log is written upon transaction commit, which means that applications will wait on the transaction log IO to complete before the commit can return control. To minimize the IO waits, which might slow down HANA's otherwise memory-speed operations, the redo log is placed on solid state disk in SAP-certified HANA appliances.

HANA's columnar architecture includes an implementation of the write-optimized delta store pattern discussed in Chapter 6. Transactions to columnar tables are buffered in this delta store. Initially, the data is held in row-oriented format (the L1 delta). Data then moves to the L2 delta store, which is columnar in orientation but relatively lightly compressed and unsorted. Finally, the data migrates to the main column store, in which data is highly compressed and sorted.

The HANA architecture:

1) On start-up, row-based tables and selected column tables are loaded from the database files
2) Other column tables will be loaded on demand. Reads and writes to row-oriented tables can be applied directly
3) Updates to column-based tables will be buffered in the delta store,
4) initially to the L1 row-oriented store. Data in the L1 store will consequently be promoted to the L2 store, and finally to the column store itself
5) Queries against column tables must read from both the delta store and the main column store.
6) Images of memory are copied to save points periodically
7) The save points are merged with the data files in due course (7).
8) When a commit occurs, a transaction record is written to the redo log (8) (on fast SSD).



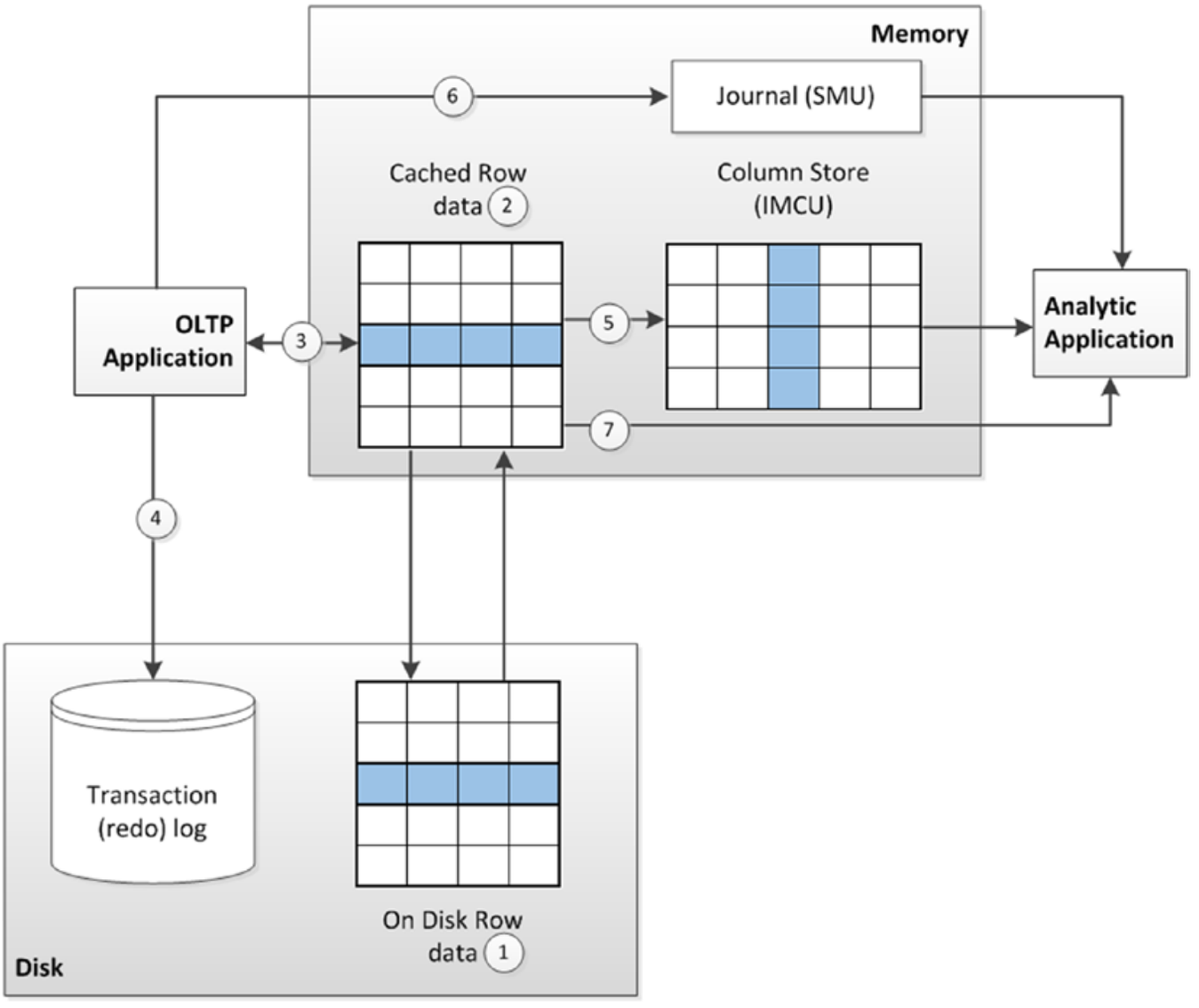
ORACLE 12C

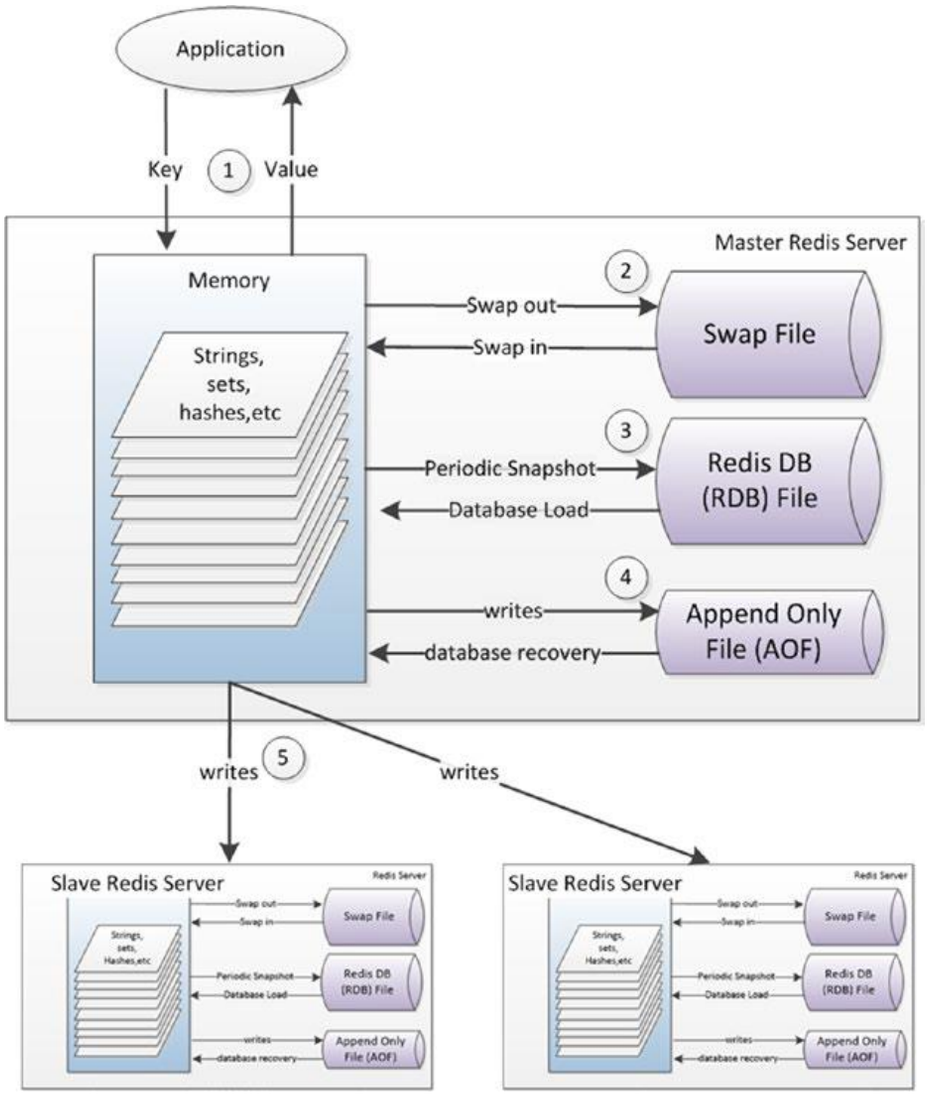
Oracle RDBMS version 12.1 introduced the “Oracle database in-memory” feature. This wording is potentially misleading, since the database as a whole is not held in memory. Rather, Oracle has implemented an in-memory column store to supplement its disk-based row store.

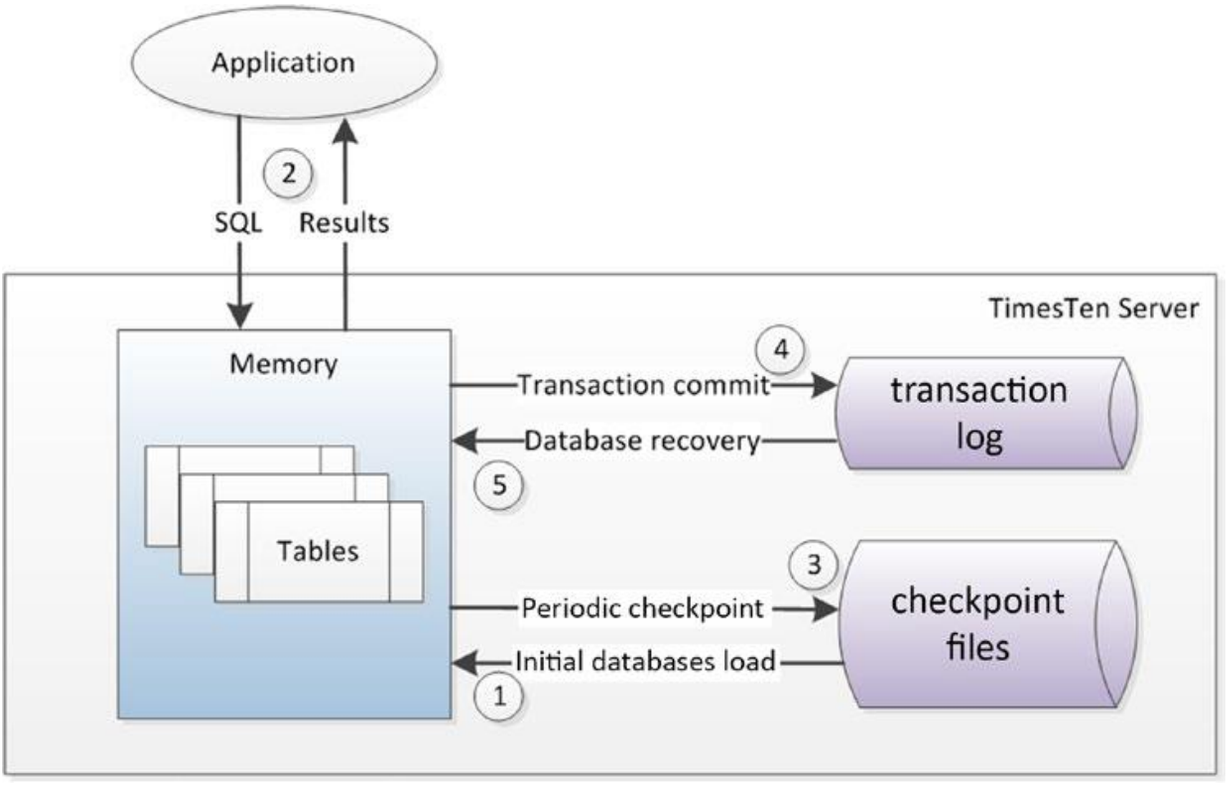
The Oracle in-memory column store architecture

1) OLTP applications work with the database in the usual manner. Data is maintained in disk files
2) but cached in memory
3) An OLTP application primarily reads and writes from memory
4) but any committed transactions are written immediately to the transaction log on disk

5) When required or as configured, row data is loaded into a columnar representation for use by analytic applications
6) Any transactions that are committed once the data is loaded into columnar format are recorded in a journal
7) and analytic queries will consult the journal to determine if they need to read updated data from the row store or possibly rebuild the columnar structure.




Application
Key
1
Value
Master Redis Server
Memory
Strings,
sets,
hashes,etc
2
Swap out
Swap in
Swap File
3
Periodic Snapshot
Database Load
Redis DB
(RDB) File
4
writes
database recovery
Append Only
File (AOF)
writes
5
writes
Slave Redis Server
Slave Redis Server


Application
2
SQL
Results
TimesTen Server
Memory
4
Transaction commit
transaction log
Database recovery
5
Tables
3
Periodic checkpoint
checkpoint files
Initial databases load
1

TimesTen implements a fairly familiar SQL-based relational model.

All data is memory resident.

Persistence is achieved by writing:

• periodic snapshots of memory to disk

• transaction log following transaction commits.

By default, the disk writes are asynchronous! To ensure ACID transactions the database can be configured for synchronous writes to a transaction log after each write (performance loss).