[]pgf@stop

SICHUAN UNIVERSITY

Database System Concepts

Database-System Architectures

伍元凯

College of Computer Science (Software), Sichuan University

wuyk0@scu.edu.cn

2023/06/07

Key-value store

A simple data storage system that stores data as a collection of key-value pairs. In this data model, each piece of data is associated with a unique key, allowing for efficient retrieval and storage.

A distributed key-value store is built to run on multiple computers working together, and thus allows you to work with larger data sets because more servers with more memory now hold the data. By distributing the store across multiple servers, you can **increase processing performance**. And if you leverage replication in your distributed key-value store, you **increase its fault tolerance**.

Toy example

```
1 class KeyValueStore:
2     def __init__(self):
3         self.data = {}
4     def set(self, key, value):
5         self.data[key] = value
6     def get(self, key):
7         return self.data.get(key)
8     def delete(self, key):
9         if key in self.data:
10            del self.data[key]
```

```
1 store = KeyValueStore()
2 store.set('name', 'John')
3 store.set('age', 25)
4 print(store.get('name'))  # Output: John
5 print(store.get('age'))   # Output: 25
6 store.delete('age')
7 print(store.get('age'))   # Output: None (age key-value pair was
      deleted)
```

horizontal partitioning

In a parallel storage system, the tuples of a relation are partitioned (divided) among many nodes, so that each tuple resides on one node.

vertical partitioning

A relation $r(A, B, C, D)$ where A is a primary key, may be vertically partitioned into $r(A, B)$ and $r(A, C, D)$, if many queries require B values, while C and D values are large in size and not required for many queries.

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
1	Alice	Anderson	Austin
2	Bob	Best	Boston
3	Carrie	Conway	Chicago
4	David	Doe	Denver

Vertical Shards

VS1

CUSTOMER ID	FIRST NAME	LAST NAME
1	Alice	Anderson
2	Bob	Best
3	Carrie	Conway
4	David	Doe

VS2

CUSTOMER ID	CITY
1	Austin
2	Boston
3	Chicago
4	Denver

Horizontal Shards

HS1

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
1	Alice	Anderson	Austin
2	Bob	Best	Boston

HS2

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
3	Carrie	Conway	Chicago
4	David	Doe	Denver

Partitioning

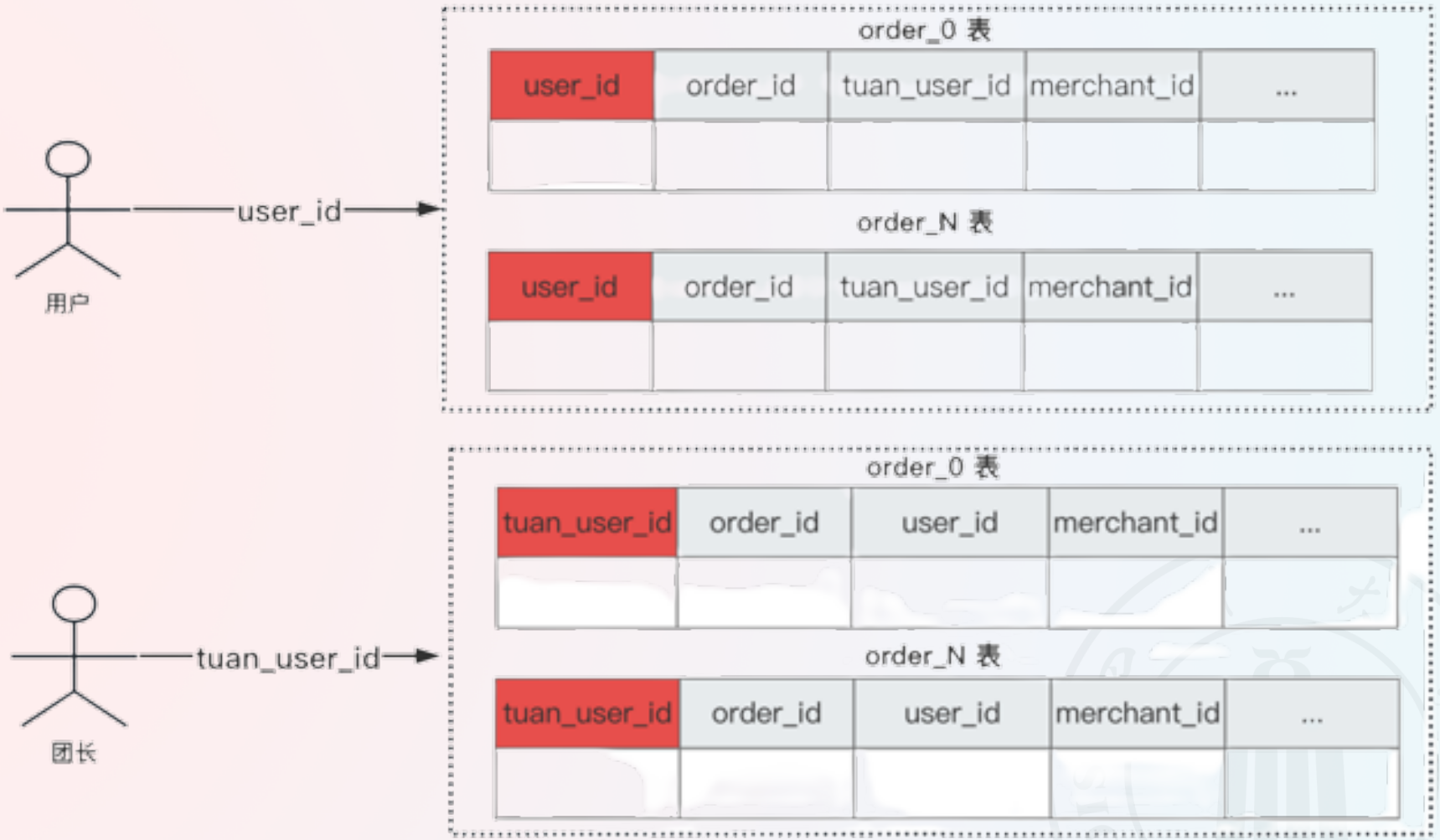
The partitioning of tuples of a relation r into multiple physical relations $r_1, r_2, \ldots, r_n$, where all the physical relations r_i are stored in a single node. The relation r is not stored, but treated as a view defined by the query $r_1 \cup r_2 \cup \ldots \cup r_n$.

Intra-node partitioning

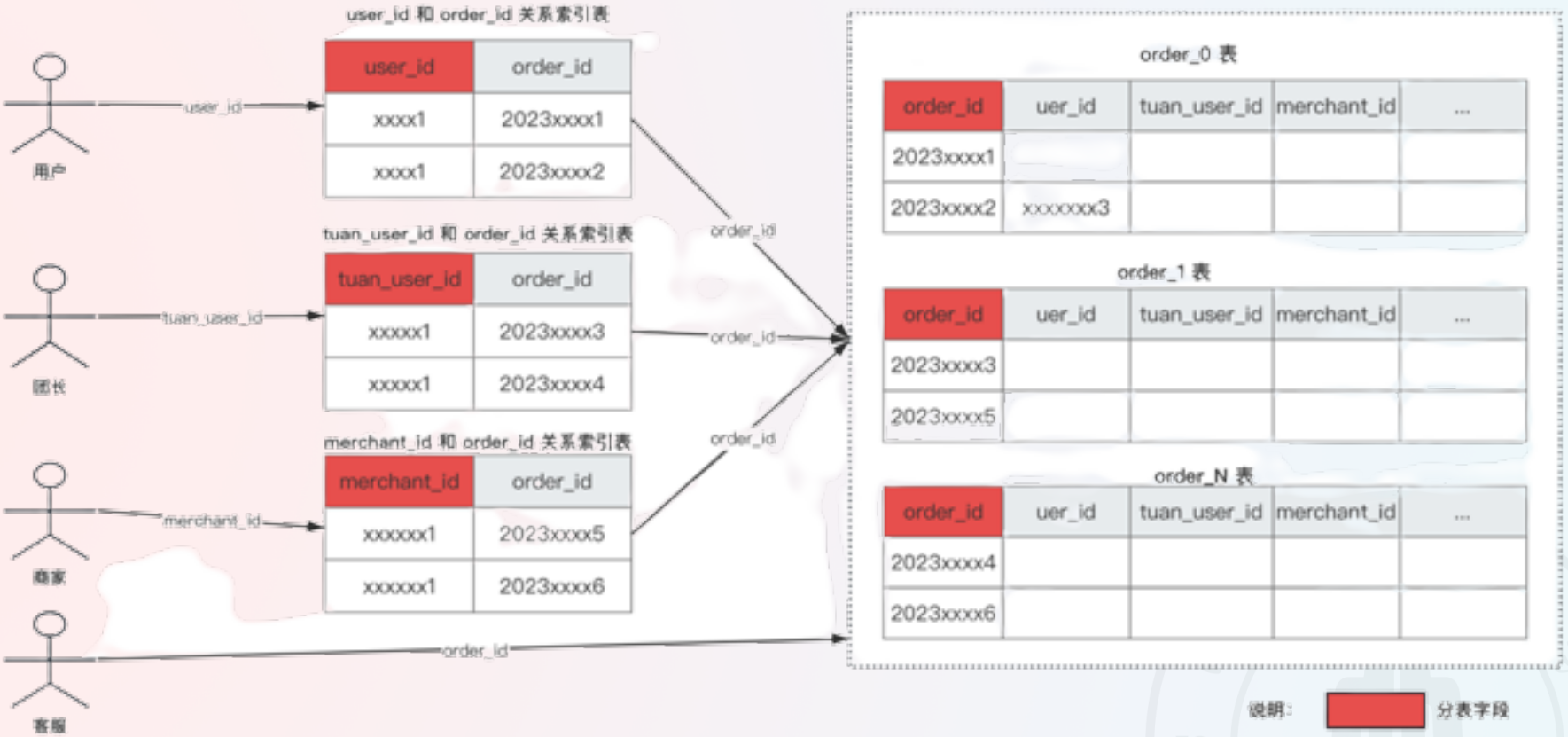
Frequently accessed tuples are stored separately from infrequently accessed tuples.

Example

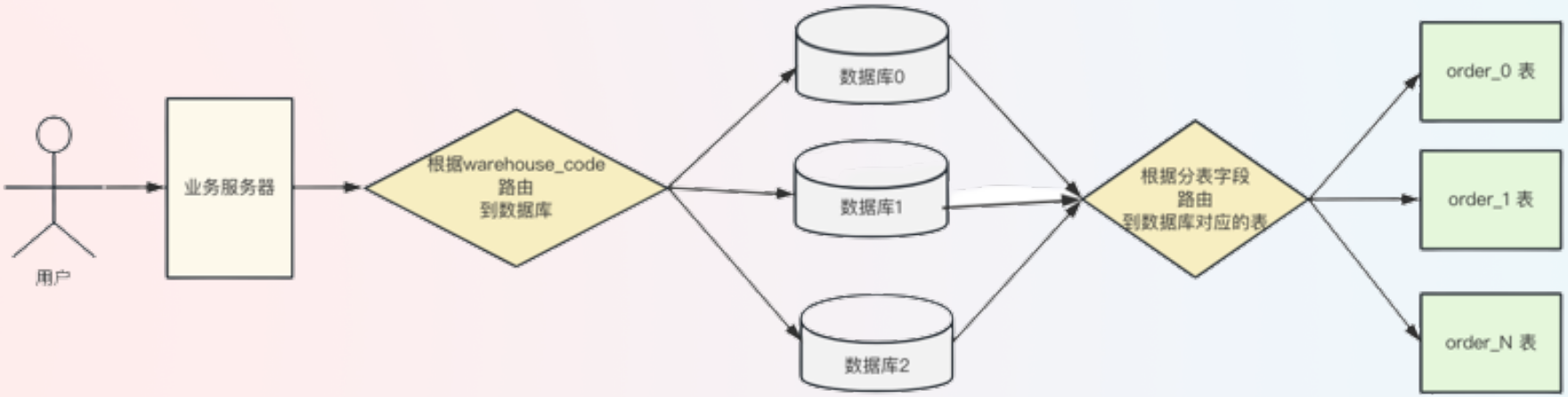
- **User Perspective**: Querying all of their own orders, therefore, the orders can be partitioned using `user_id`, placing all orders of a particular user in the same table.
- **Group Leader Perspective**: Querying all orders placed by users under their leadership, therefore, the orders can be partitioned using `tuan_user_id`, placing all orders of a particular group leader in the same table.
- **Merchant Perspective**: Querying all orders placed by users for their business, therefore, the orders can be partitioned using `merchant_id`, placing all orders of a particular merchant in the same table.
- **Customer Service Perspective**: Querying a specific order by its order number, therefore, partitioning the orders using `order_id` allows for quick retrieval of order information.



Redundant attribute (sacrifice space)



Relation index table (sacrifice time)

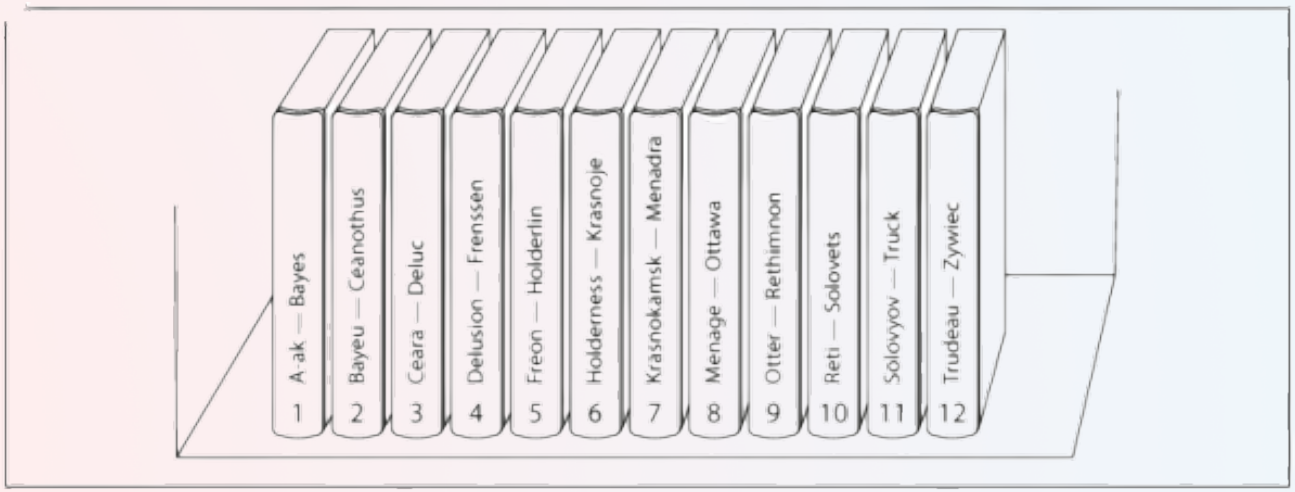


Therefore, the final database sharding and partitioning model is based on the warehouse code, using the warehouse code to determine the database shards, and routing based on the partitioning key to the order table.

https://www.yuanjava.cn/posts/mysql-subdatabase-subtable/

目录

- Round-robin: This strategy scans the relation in any order and sends the ith tuple fetched during the scan to node number $N_{((i-1) \mod n)+1}$. The round-robin scheme ensures an even distribution of tuples across nodes; that is, each node has approximately the same number of tuples as the others.
- Hash partitioning: A hash function is chosen whose range is $1, 2, \ldots, n$. Each tuple of the original relation is hashed on the partitioning attributes. If the hash function returns i, then the tuple is placed on node N_i.
- Range partitioning: This strategy distributes tuples by assigning contiguous attribute-value ranges to each node. It chooses a partitioning attribute, A, and a partitioning vector $[v_1, v_2, \ldots, v_{n-1}]$, such that if $i < j$, then $v_i < v_j$. The relation is partitioned as follows: Consider a tuple t such that $t[A] = x$. If $x < v_1$, then t goes on node N_1. If $x \geq v_{n-1}$, then t goes on node N_n. If $v_i \leq x < v_{i+1}$, then t goes on node N_{i+1}.



range partitioning

The ranges of keys are not necessarily evenly spaced, because your data may not be evenly distributed. For example, volume 1 contains words starting with A and B, but volume 12 contains words starting with T, U, V, W, X, Y, and Z. Simply having one volume per two letters of the alphabet would lead to some volumes being much bigger than others.

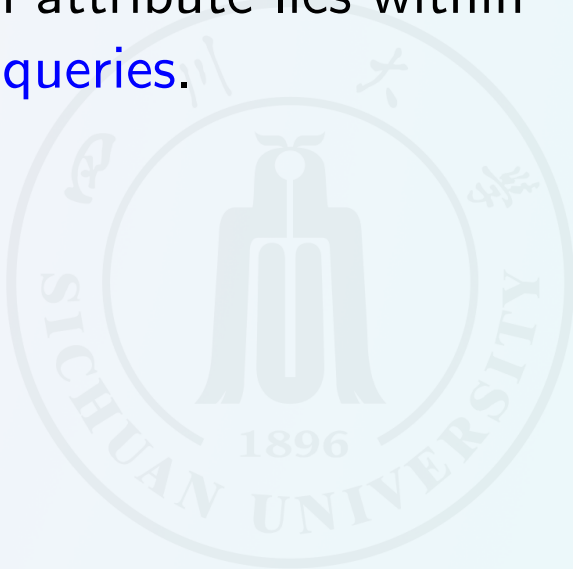
How partitioning is maintained when a relation is updated:

1. When a tuple is inserted into a relation, it is sent to the appropriate node based on the partitioning strategy
2. If a tuple is deleted, its location is first found based on the value of its partitioning attribute (for round-robin, all partitions are searched). The tuple is then deleted from wherever it is located.
3. If a tuple is updated, its location is not affected if either round-robin partitioning is used or if the update does not affect a partitioning attribute. However, if range partitioning or hash partitioning is used, and the update affects a partitioning attribute, the location of the tuple may be affected. In this case:
   - The original tuple is deleted from the original location
   - The updated tuple is inserted and sent to the appropriate node based on the partitioning strategy used.

目录

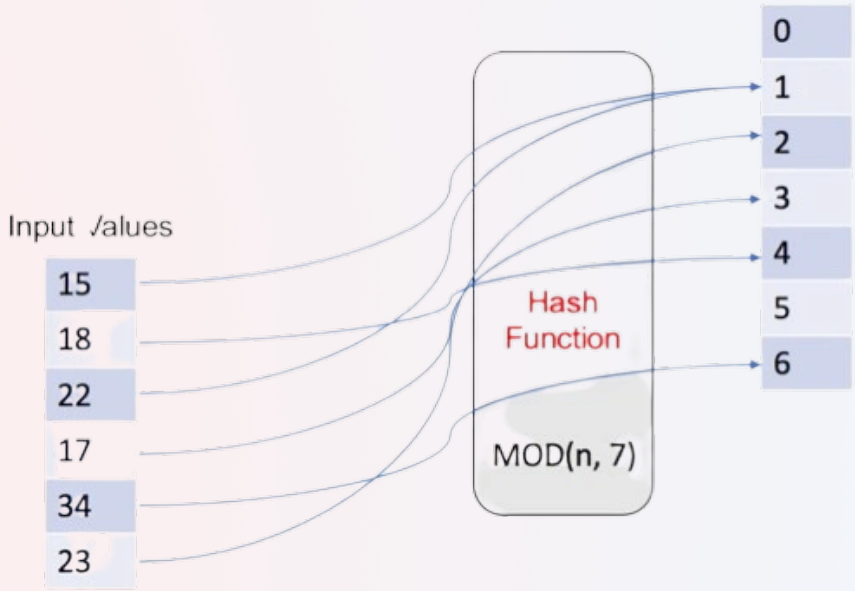
Access to data can be classified as follows:

- Scanning the entire relation.
- Locating a tuple associatively; these queries, called point queries, seek tuples that have a specified value for a specific attribute.
- Locating all tuples for which the value of a given attribute lies within a specified range; these queries are called range queries.



The different partitioning techniques support these types of access at different levels of efficiency:

- **Round-robin**. The scheme is ideally suited for applications that wish to read the entire relation sequentially for each query.
- **Hash partitioning**. This scheme is best suited for point queries based on the partitioning attribute. Hash partitioning is also useful for sequential scans of the entire relation. If the hash function is a good randomizing function, and the partitioning attributes form a key of the relation, then the number of tuples in each of the nodes is approximately the same.
- **Range partitioning**. This scheme is well suited for point and range queries on the partitioning attribute.



Basic principle of a hash function

Hash-based partitioning is also not well suited for answering range queries, since, typically, hash functions do not preserve proximity within a range. Therefore, all the nodes need to be scanned for range queries to be answered.

Execution skew

if there are many tuples in the queried range (as there are when the queried range is a larger fraction of the domain of the relation), many tuples have to be retrieved from a few nodes, resulting in an I/O bottleneck (hot spot) at those nodes.
All processing occurs in one—or only a few—partitions.

Partitioning is important for large relations. Large databases that benefit from parallel storage often have some small relations. **Partitioning is not a good idea for such small relations, since each node would end up with just a few tuples**.

Data distribution skew may be caused by one of two factors

- Attribute-value skew, which refers to the fact that some values appear in the partitioning attributes of many tuples.
- Partition skew, which refers to the fact that there may be load imbalance in the
- there may be execution skew even if there is no skew in the distribution of tuples, if queries tend to access some partitions more often than others. partitioning, even when there is no attribute skew.

例 1

1000 tuples is divided into 10 parts. if even **one partition** happens to be of size 200, the speedup that we would obtain by accessing the partitions in parallel is only **5**

例 2

If the same relation has to be partitioned into 100 parts, even one partition has 40 tuples the speedup that we would obtain by accessing them in parallel **would be 25, rather than 100**.

Data distribution skew in range partitioning can be avoided by choosing a balanced range-partitioning vector, which evenly distributes tuples across all nodes.

A balanced range-partitioning vector can be constructed by sorting, as follows: The relation is first sorted on the partitioning attributes. The relation is then scanned in sorted order. After every 1⊠n of the relation has been read, the value of the partitioning attribute of the next tuple is added to the partition vector. Here, n denotes the number of partitions to be constructed (**extra I/O overhead incurred in doing the initial sort**).

The I/O overhead for constructing balanced range-partitioning vectors can be reduced by using a precomputed frequency table, or histogram, of the attribute values for each attribute of each relation

例 3

$$(4, 8, 14, 19) \tag{1}$$

The histogram indicates that 1/5th of the tuples have age less than 4, another 1/5th have age ≥ 4 but < 8, and so on, with the last 1/5th having age ≥ 19.

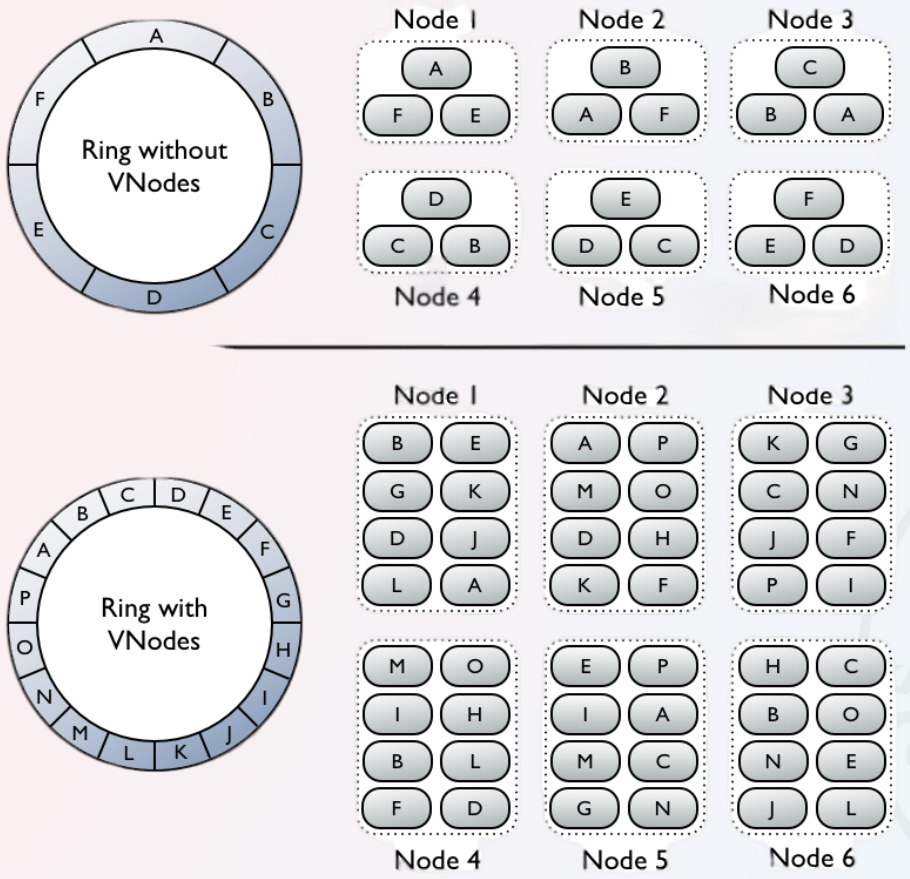
Drawback (static)

The partitioning is decided at some point and is not automatically updated as tuples are inserted, deleted, or updated. The partitioning vectors can be recomputed, and the data repartitioned, whenever the system detects skew in data distribution.
However, the cost of repartitioning can be quite large, and doing it periodically would introduce a high load which can affect normal processing.

目录

virtual nodes

In the virtual nodes approach, we pretend there are several times as many virtual nodes as the number



virtual nodes illustration

例 4

One way to map virtual nodes to real nodes is round-robin allocation; thus, if there are n real nodes numbered 1 to n, virtual node i is mapped to real node $((i-1) mod n)+1$. The idea is that even if one range had many more tuples than the others because of skew, these tuples would get split across multiple virtual nodes ranges.

The system must then record this mapping and use it to route accesses to the correct real node.

elasticity of storage

When a new node is added, some of the virtual nodes are migrated to the new real node, which can be done without affecting any of the other virtual nodes. If the amount of data mapped to each virtual node is small, the migration of a virtual node from one node to another can be done relatively fast, minimizing disruption.

Repartitioning

目录

Dynamic repartitioning can be done in an efficient manner by instead exploiting the virtual node scheme. The basic idea is to split a virtual node into two virtual nodes when it has too many tuples, or too much load

例 5

if the virtual node corresponding to a range of timestamps 2017-01-01 to MaxDate were to become overfull, the partition could be split into two partitions. For example, if half the tuples in this range have timestamps less than 2018-01-01, one partition would have timestamps from 2017-01-01 to less than 2018-01-01, and the other would have tuples with timestamps from 2018-01-01 to MaxDate

In data storage systems, the term table refers to a collection of data items. Tables are partitioned into multiple tablets.
The system needs to maintain a partition table, which provides a mapping from the partitioning key ranges to a tablet identifier, as well as the real node on which the tablet data reside.

Value	Tablet ID	Node ID
2012-01-01	Tablet0	Node0
2013-01-01	Tablet1	Node1
2014-01-01	Tablet2	Node2
2015-01-01	Tablet3	Node2
2016-01-01	Tablet4	Node0
2017-01-01	Tablet5	Node1
MaxDate	Tablet6	Node1

Tablet and Node Mapping

Most parallel data storage systems store the partition table at a master node. However, to support a large number of requests each second, the partition table is usually replicated, either to all client nodes that access data or to multiple routers.

Value	Tablet ID	Node ID
2012-01-01	Tablet0	Node0
2013-01-01	Tablet1	Node0
2014-01-01	Tablet2	Node2
2015-01-01	Tablet3	Node2
2016-01-01	Tablet4	Node0
2017-01-01	Tablet5	Node1
2018-01-01	Tablet6	Node1
MaxDate	Tablet7	Node1

Example partition table after tablet split and tablet move

With a large number of nodes, the probability that at least one node will malfunction in a parallel system is significantly greater than in a single-node system.

例 6

if a single node would fail once **every 5 years**, a system with 100 nodes would have a failure **every 18 days**.

To ensure tuples are not lost on node failure, tuples are replicated across at least two nodes, and often three nodes.

Value	Tablet ID	Node ID
2012-01-01	Tablet0	Node0,Node1
2013-01-01	Tablet1	Node0,Node2
2014-01-01	Tablet2	Node2,Node0
2015-01-01	Tablet3	Node2,Node1
2016-01-01	Tablet4	Node0,Node1
2017-01-01	Tablet5	Node1,Node0
2018-01-01	Tablet6	Node1,Node2
MaxDate	Tablet7	Node1,Node2

Partition table with replication

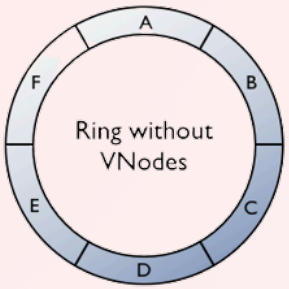
The location of the nodes where the replicas of a partition are stored must be chosen carefully

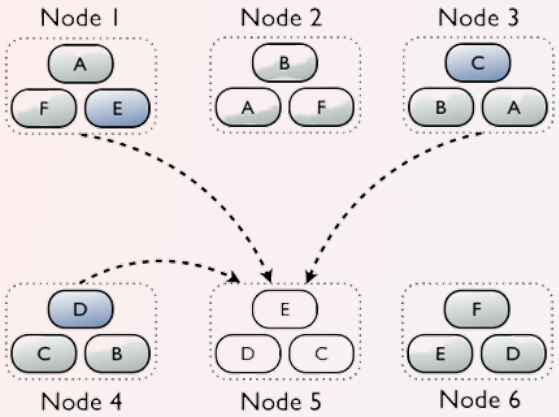
- Replication within a data center: Replication to another node within the same rack as the first node reduces network demand on the network between racks (**With the tree-like interconnection topology**).
- Replication across data centers: To deal with the possibility of failure of an entire data center, partitions may also be replicated at one or more geographically separated data centers

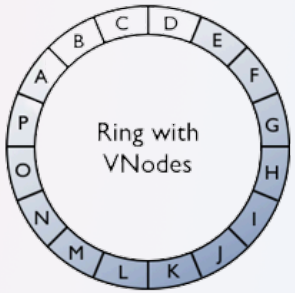
例 7

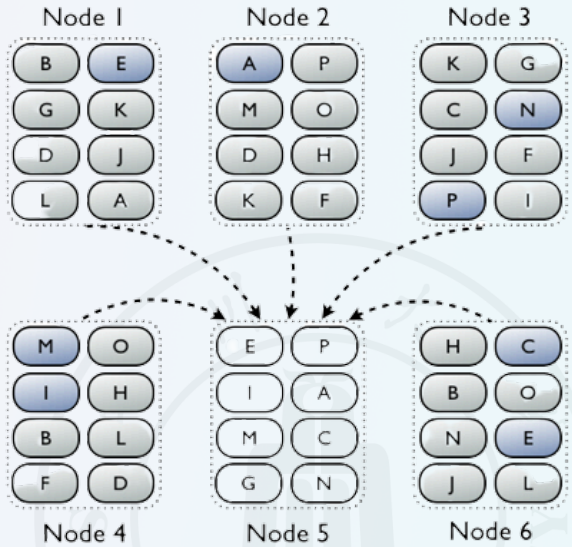
Suppose all the partitions at a node N_1 are replicated at a single node N_2, and N_1 fails. Then, node N_2 will have to handle all the requests that would originally have gone to N_1, as well as requests routed to node N_2. As a result, node N_2 would have to perform twice as much work as other nodes in the system, resulting in execution skew during failure of node N_1.

To avoid this problem, the replicas of partitions residing at a node, say N1, are spread across multiple other nodes.









目录

One way of ensuring that reads get the latest value is to treat one of the replicas of each partition as a master replica. All updates are sent to the master replica and are then propagated to other replicas. Reads are also sent to the master replica, so that reads get the latest version of any data item even if updates have not yet been applied to the other replicas

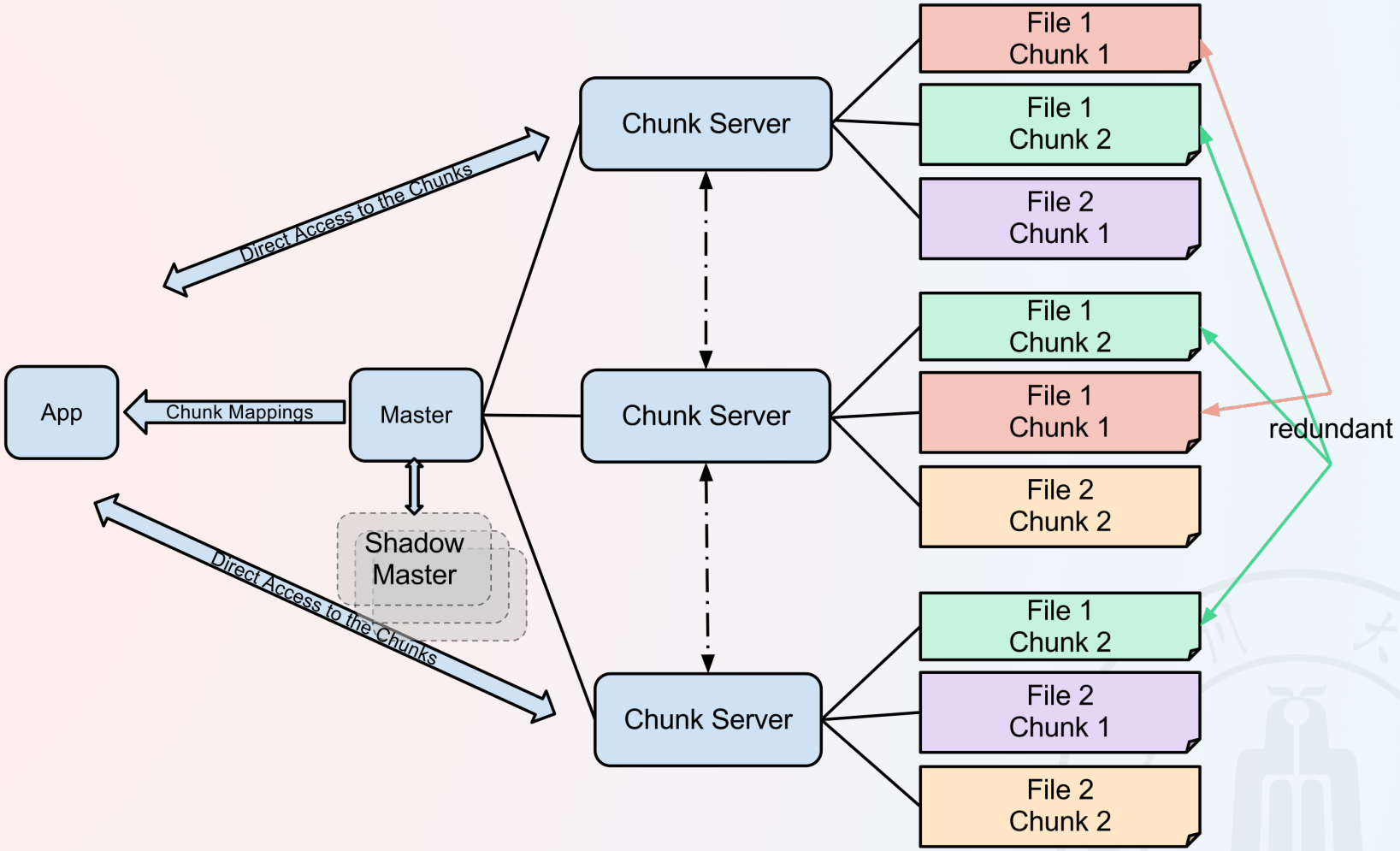
Three solutions are commonly used to update replicas

- **The two-phase commit (2PC)** protocol, which ensures that multiple updates performed by a transaction are applied atomically across multiple sites

- **Persistent messaging systems** can be used to update replicas as follows: An update to a tuple is registered as a persistent message, sent to all replicas of the tuple. Once the message is recorded, the persistent messaging system ensures it will be delivered to all replicas. Thus, all replicas will get the update, eventually; the property is known as eventual consistency of replicas.
- Protocols called **consensus protocols**, that allow updates of replicas to proceed even in the face of failures, when some replicas may not be reachable, can be used to manage update of replicas.

A distributed file system stores files across a large collection of machines while giving a single-file-system view to clients. As with any file system, there is a system of file names and directories, which clients can use to identify and access files. Clients do not need to bother about where the files are stored.

A landmark system in this context was the Google File System (GFS), developed in the early 2000s



Google File System is designed for system-to-system interaction, and not for user-to-system interaction. The chunk servers replicate the data automatically.

File systems typically support two kinds of **metadata**:

- A directory system, which allows a hierarchical organization of files into directories and subdirectories
- A mapping from a file name to the sequence of identifiers of blocks that store the actual data in each file

Hadoop File System (HDFS):

- The nodes (machines) which store data blocks in HDFS are called datanodes.
- GFS and HDFS store the file system metadata at a single node, called the namenode in HDFS.

A key-value store provides a way to store or update a data item (value) with an associated key and to retrieve the data item with a given key.

Many key-value stores support some form of flexible schema.

- Some allow column names to be specified as part of a schema definition, similar to relational data stores
- Others allow columns to be added to, or deleted from, individual tuples; such key-value stores are sometimes referred to as wide-column store
- t other key-value stores allow the value stored with a key to have a complex structure, typically based on JSON; they are sometimes referred to as document stores.

Data Representation | 目录

Many key-value stores support the **JavaScript Object Notation (JSON)** representation

```
{
  "name": "John Doe",
  "age": 30,
  "city": "New York"
}

{
  "fruits": ["apple", "banana", "orange"],
  "vegetables": ["carrot", "broccoli"]
}
```

In **Bigtable**, a record is not stored as a single value but is instead split into component attributes that are stored separately. Thus, the key for an attribute value conceptually consists of (record-identifier, attribute-name).

```
Row Key: product123

Columns:
  - columnFamily:info, columnQualifier:name, value: "Smartphone"
  - columnFamily:info, columnQualifier:brand, value: "ABC
      Electronics"
  - columnFamily:info, columnQualifier:price, value: "499.99"
  - columnFamily:details, columnQualifier:color, value: "Black"
  - columnFamily:details, columnQualifier:screenSize, value: "6
      inches"
```

```
Get("product123", "info", "name")
```

目录

Storing and Retrieving Data

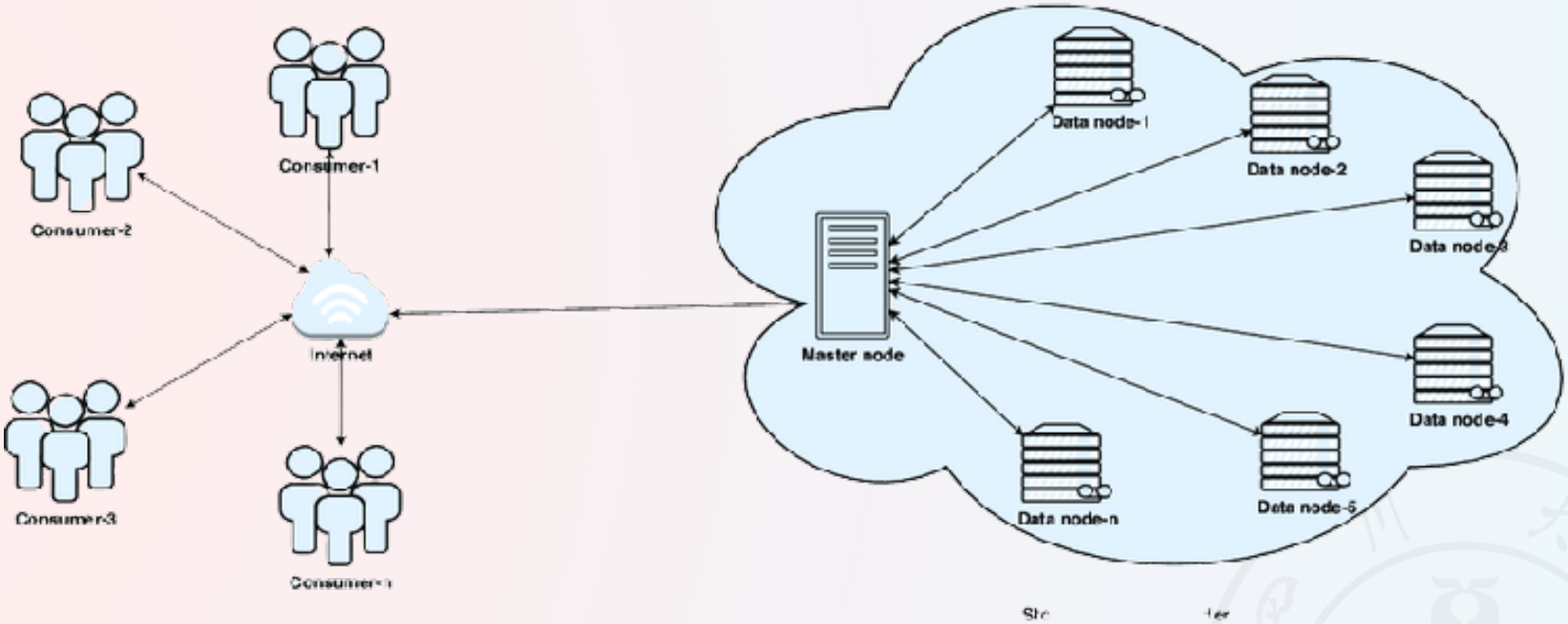
Tablet server

The node that acts as the server for a particular tablet; all requests related to a tablet are sent to the tablet server for that tablet

Master

A site that stores a master copy of the partition information, including, for each tablet, the key ranges for the tablet, the sites storing the replicas of the tablet, and the current tablet server for that tablet

- By replicating the partition information to the client sites; the key-value store API used by clients looks up the partition information copy stored at the client to decide where to route a request.
- By replicating the partition information to a set of router sites, which route requests to the site with the appropriate tablet. Requests can be sent to any one of the router sites, which forward the request to the correct tablet master.



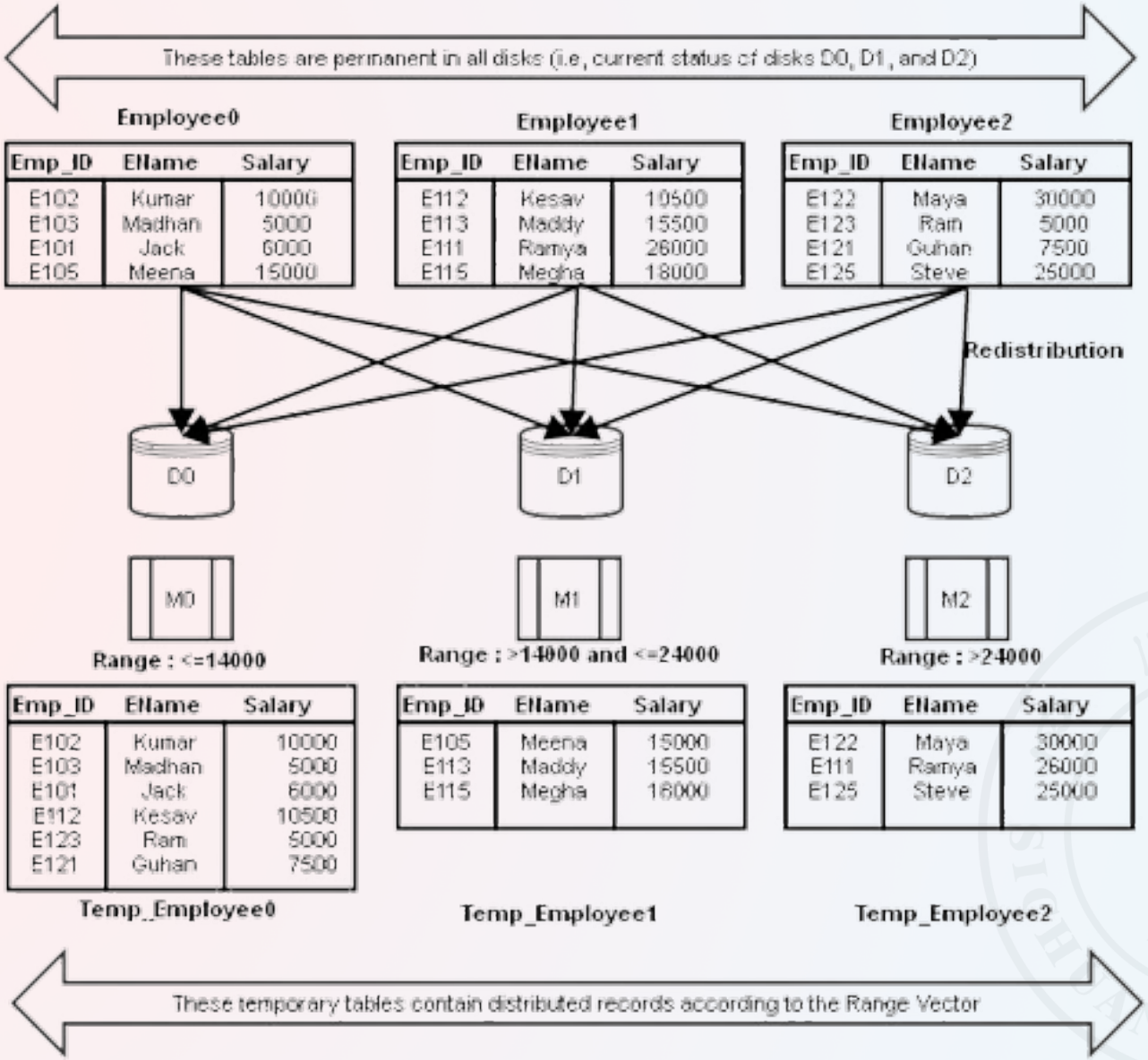
Master and data node in cloud storage architecture.

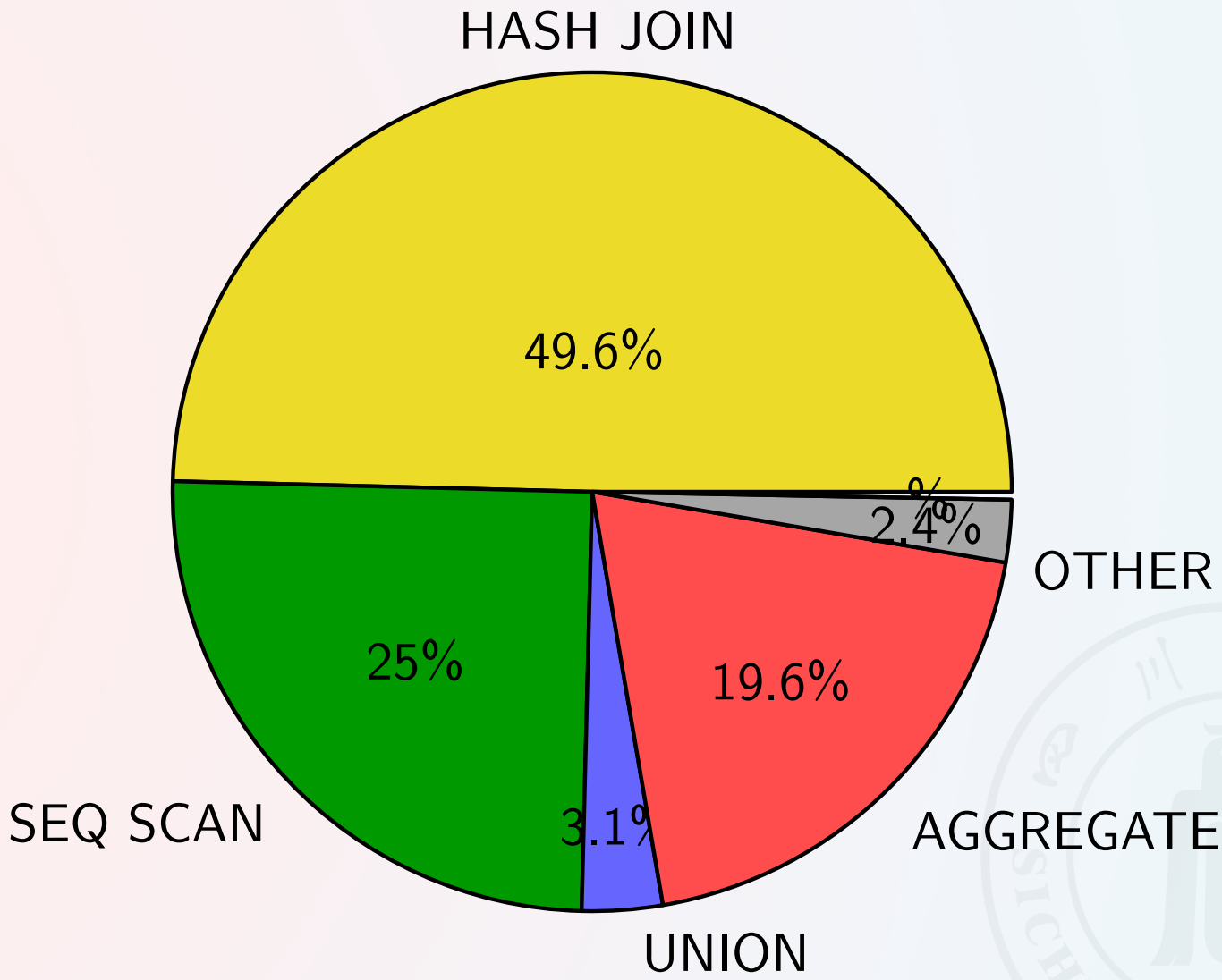
One of the key motivations for geographic distribution is fault tolerance, which allows the system to continue functioning even if an entire data center fails due to a disaster such as a fire or an earthquake; in fact, earthquakes could cause all data centers in a region to fail.

A key performance issue with geographical replication of data is that the latency across geographical regions is much higher than the latency within a data center.

Suppose that we choose nodes $N1, N2, \ldots, Nm$ to sort the relation. There are two steps involved in this operation:

1. Redistribute the tuples in the relation, using a range-partition strategy, so that all tuples that lie within the ith range are sent to node ni, which stores the relation temporarily on its local disk
2. Each of the nodes sorts its partition of the relation locally, without interaction with the other nodes. Each node executes the same operation—namely, sorting—on a different data set.


These tables are permanent in all disks (i.e, current status of disks D0, D1, and D2)
Employee0
Emp_ID EName Salary
E102 Kumar 10000
E103 Madhan 5000
E101 Jack 6000
E105 Meena 15000
Employee1
Emp_ID EName Salary
E112 Kesav 10500
E113 Maddy 15500
E111 Ramya 26000
E115 Megha 18000
Employee2
Emp_ID EName Salary
E122 Maya 30000
E123 Ram 5000
E121 Guhan 7500
E125 Steve 25000
Redistribution
D0
D1
D2
M0
M1
M2
Range : <=14000
Range : >14000 and <=24000
Range : >24000
Emp_ID EName Salary
E102 Kumar 10000
E103 Madhan 5000
E101 Jack 6000
E112 Kesav 10500
E123 Ram 5000
E121 Guhan 7500
Temp_Employee0
Emp_ID EName Salary
E105 Meena 15000
E113 Maddy 15500
E115 Megha 16000
Temp_Employee1
Emp_ID EName Salary
E122 Maya 30000
E111 Ramya 26000
E125 Steve 25000
Temp_Employee2
These temporary tables contain distributed records according to the Range Vector



% of Total CPU Time Spent in Query Operators - Workload: TPC-H Benchmark

Partitioned join

The system partitions the relations r and s each into m partitions, denoted $r_1, r_2, \ldots, r_m$ and $s_1, s_2, \ldots, s_m$. In a partitioned join, however, there are two different ways of partitioning r and s:

Specifically, each node N_i reads in the tuples of one of the relations, say r, from local disk, computes for each tuple t the partition r_j to which t belongs, and sends the tuple t to node N_j. Each node also simultaneously receives tuples that are sent to it and stores them on its local disk. The process is repeated for all tuples from the other relation, s.

1. **Partition**: Divide the tuples of R and S into sets using a hash on the join key.
2. **Build**: Scan relation R and create a hash table on join key.
3. **Probe**: For each tuple in S, look up its join key in hash table for R. If a match is found, output combined tuple.

Partitioning is not applicable to all types of joins. For instance, if the join condition is an inequality, such as $r \bowtie r.a < s.b$, it is possible that all tuples in r join with some tuple in s (and vice versa).

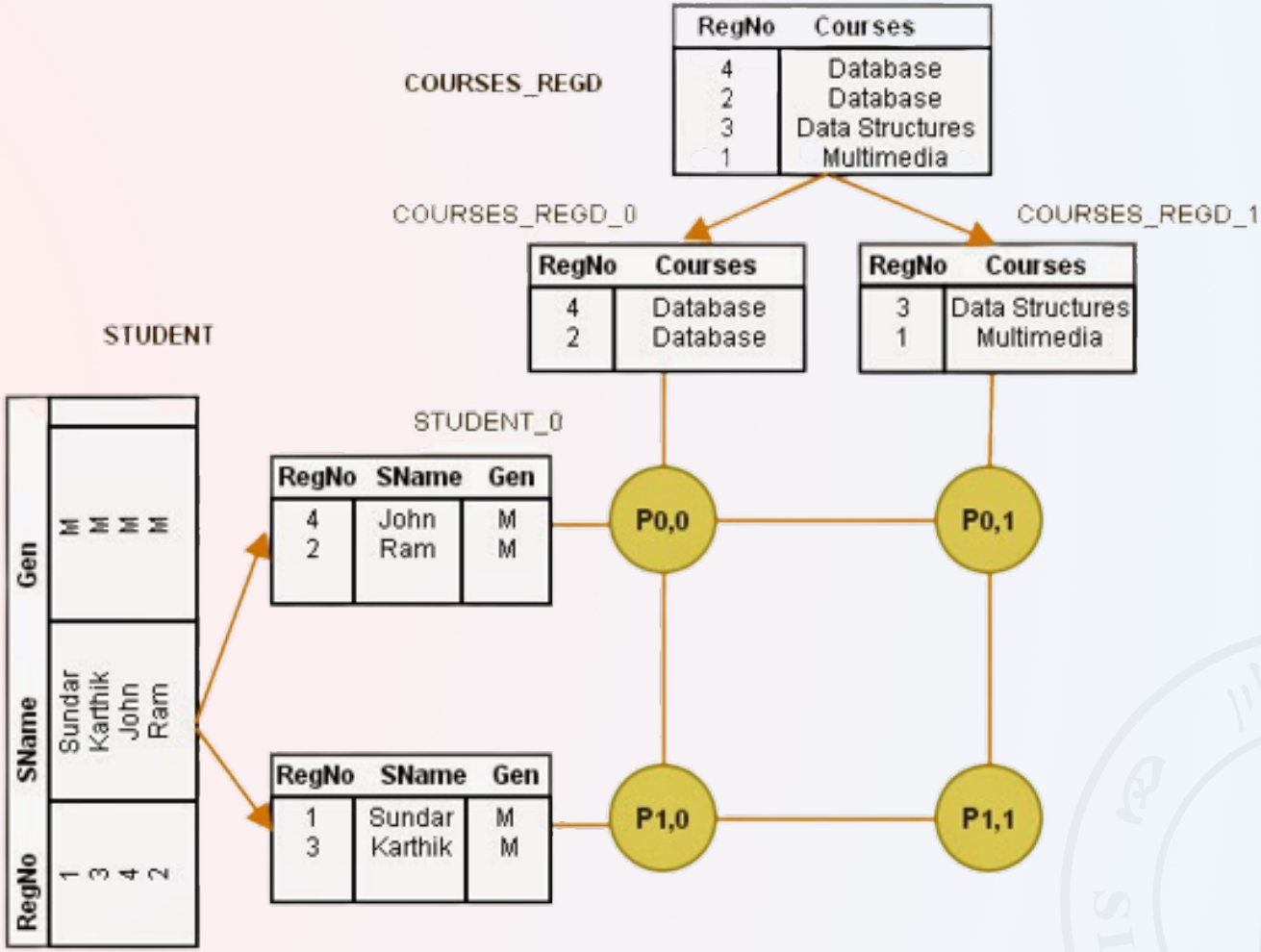
asymmetric fragment-and-replicate join

1. The system partitions one of the relations r. Any partitioning technique can be used on r, including round-robin partitioning.
2. The system replicates the other relation, s, across all the nodes.
3. Node N_i then locally computes the join of r_i with all of s, using any join technique

The asymmetric fragment-and-replicate join technique is also referred to as broadcast join.

fragment-and-replicate join

Let the nodes be $N_{1,1}$, $N_{1,2}$, ..., $N_{1,m}$, $N_{2,1}$, ..., $N_{n,m}$. Node $N_{i,j}$ computes the join of r_i with s_j. To ensure that each node $N_{i,j}$ gets all tuples of r_i and s_j, the system replicates r_i to nodes $N_{i,1}$, $N_{i,2}$, ..., $N_{i,m}$, and replicates s_j to nodes $N_{1,j}$, $N_{2,j}$, ..., $N_{n,j}$. Any join technique can be used at each node $N_{i,j}$.

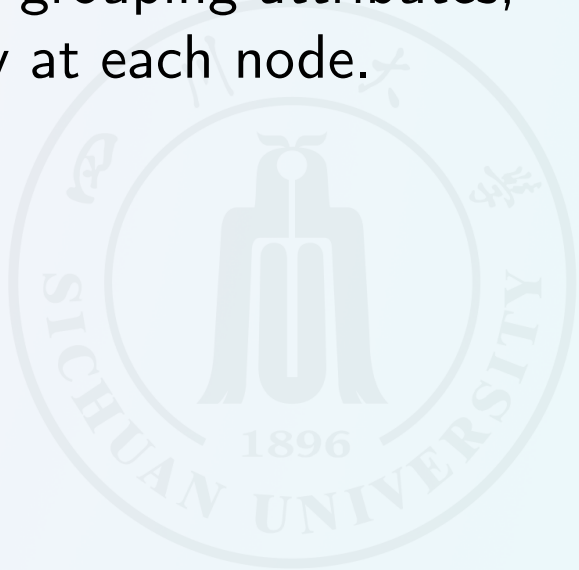


A fragment-and-replicate join example

- Hash partitioning using any good hash function usually works quite well at balancing the load across nodes, unless some join attribute values occur very frequently.
- Range partitioning, on the other hand, is more vulnerable to join skew, unless the ranges are carefully chosen to balance the load.
- Virtual-node partitioning with can help in reducing skew at the level of real nodes even if there is skew at the level of virtual nodes, since the skewed virtual nodes tend to get spread over multiple real nodes.

- **Selection**. Let the selection be $\sigma_\theta(r)$. Consider first the case where θ is of the form $a_i = v$, where a_i is an attribute and v is a value. If the relation r is partitioned on a_i, the selection proceeds at a single node. If θ is of the form $l \leq a_i \leq u$—that is, θ is a range selection—and the relation has been range-partitioned on a_i, then the selection proceeds at each node whose partition overlaps with the specified range of values. In all other cases, the selection proceeds in parallel at all the nodes.
- **Duplicate elimination**, we can also parallelize duplicate elimination by partitioning the tuples (by either range or hash partitioning) and eliminating duplicates locally at each node.

- **Projection**. Projection without duplicate elimination can be performed as tuples are read in from disk in parallel. If duplicates are to be eliminated, either of the techniques just described can be used.
- **Aggregation**. Consider an aggregation operation. We can parallelize the operation by partitioning the relation on the grouping attributes, and then computing the aggregate values locally at each node.



```sql
SELECT A, SUM(B) AS Total
FROM r
GROUP BY A;
```

例 8

The system can perform the sum aggregation at each node N_i on those r tuples stored at N_i. This computation results in tuples with partial sums at each node; the result at N_i has one tuple for each A value present in r tuples stored at N_i, with the sum of the B values of those tuples. The system then partitions the result of the local aggregation on the grouping attribute A and performs the aggregation again (on tuples with the partial sums) at each node N_i to get the final result.

Database integration can be done in several different ways:

- The federated database approach creates a common schema, called a global schema, for data from all the databases/data sources; each database has its own local schema. The task of creating a unified global schema from multiple local schemas is referred to as schema integration
- The data virtualization approach allows applications to access data from multiple databases/data sources, but it does not try to enforce a common schema.
- The external data approach allows database administrators to provide schema information about data that are stored in other databases, along with other information, such as connection and authorization information needed to access the data. Data stored in external sources that can be accessed from a database are referred to as external data. Foreign tables are views defined in a database whose actual data are stored in an external data source.

A wrapper provides a view of data stored at a data source, in a desired schema. For example, if the system has a global schema, and the local database schema is different from the global schema, a wrapper can provide a view of the data in the global schema.

The term data lake is used to refer to an architecture where data are stored in multiple data storage systems and in different formats, including in file systems, but can be queried from a single system.

目录

The first task in providing a unified view of data lies in creating a unified conceptual schema, a task that is referred to as schema integration.

Schema integration requires the creation of a global schema, which provides a unified view of data in different databases

global-as-view (GAV)

Schema integration also requires a way to define how data are mapped from the local schema representation at each database, to the global schema. This step can be done by defining views at each site which, transform data from the local schema to the global schema. Data in the global schema is then treated as the union of the global views at the individual site.

- Site s_1 which uses the relation *student1(ID, name, dept name)*, and the relation *studentCreds(ID, tot cred)*.
- Site s_2 which uses the relation *student2(ID, name, tot cred)*, and the relation *studentDept(ID, dept name)*.

```
1 create view student s1(ID, name, dept name, tot cred) as
2 select ID, name, dept name, tot cred
3 from student1, studentCreds
4 where student1.ID= studentCreds.ID;
```

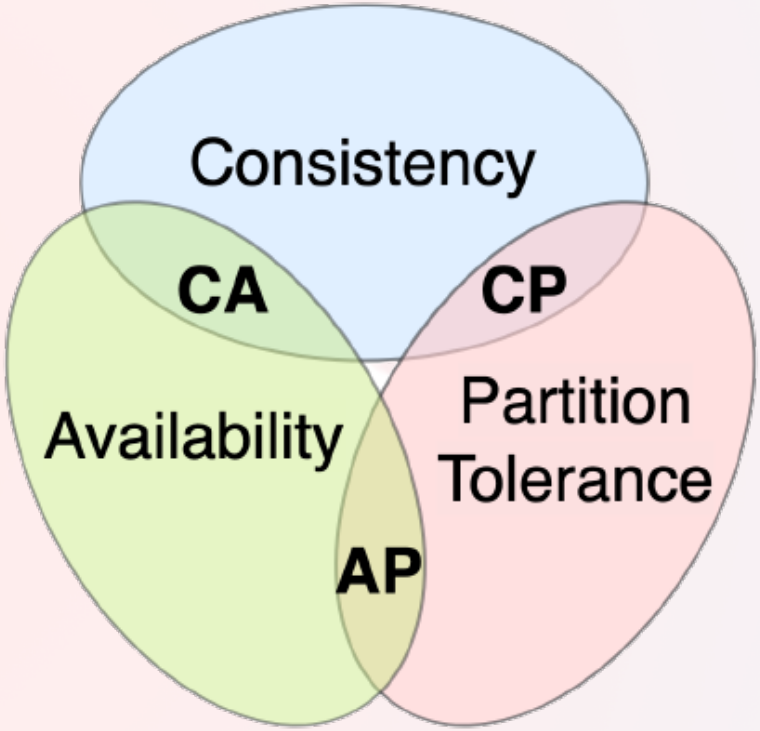
```
1 create view student s2(ID, name, dept name, tot cred) as
2 select ID, name, dept name, tot cred
3 from student2, studentDept
4 where student2.ID= studentDept.ID;
```

There are more complex mapping schemes that are designed to deal with duplication of information across sites and to allow translation of updates on the global schema into updates on the local schema. The local-as-view (LAV) approach, which defines local data in each site as a view on a conceptual unified global relation, is one such approach.

例 9

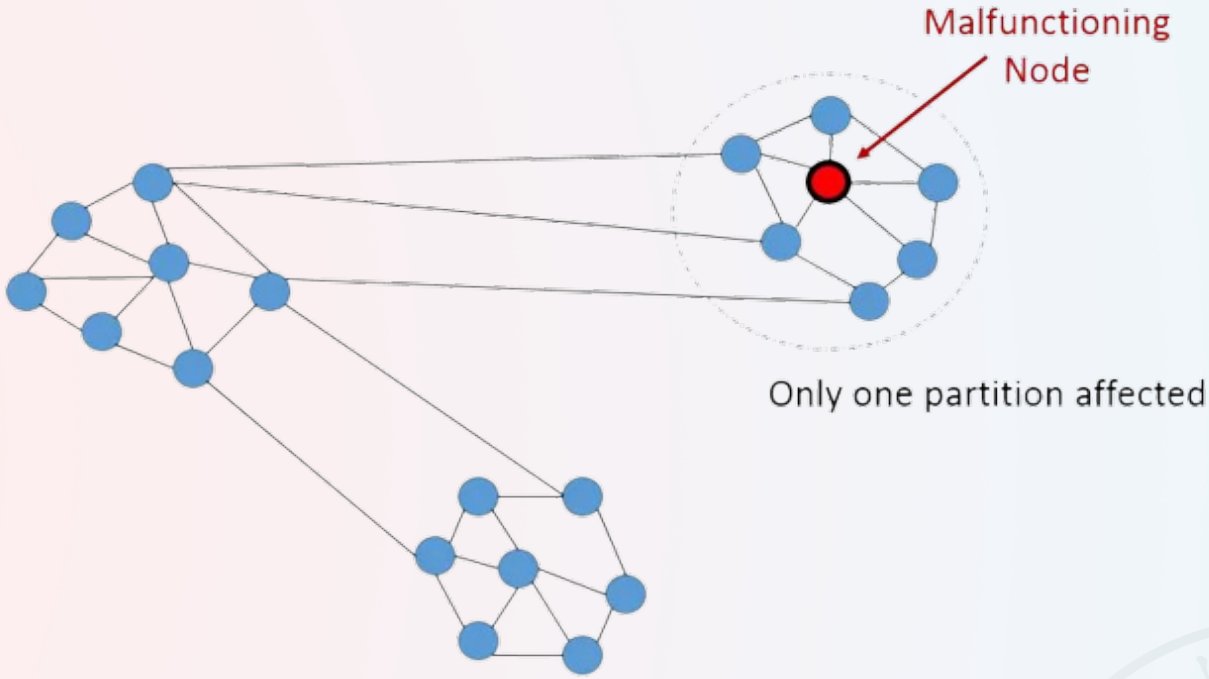
Consider for example a situation where the student relation is partitioned between two sites based on the dept name attribute, with all students in the "Comp. Sci." department at site s3 and all students in other departments in site s4

```
1 create view student s3 as
2 select *
3 from student
4 where student.dept name = 'Comp. Sci.';
5
6 create view student s4 as
7 select *
8 from student
9 where student.dept name ! = 'Comp. Sci.';
```

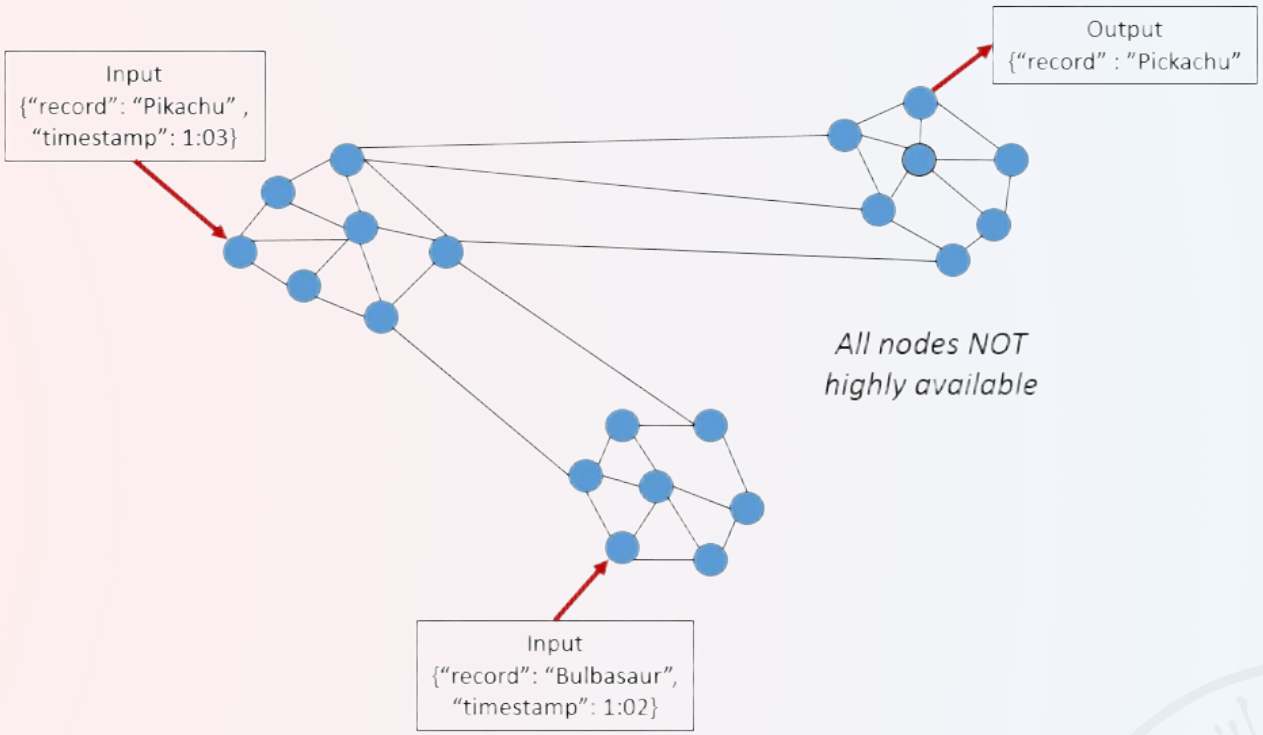


CAP Theorem is a concept that a distributed database system can only have 2 of the 3: Consistency, Availability and Partition Tolerance.

- **Consistency**. All reads receive the most recent write or an error.
- **Availability**. All reads contain data, but it might not be the most recent.
- **Partition tolerance**. The system continues to operate despite network failures (ie; dropped partitions, slow network connections, or unavailable network connections between nodes.)



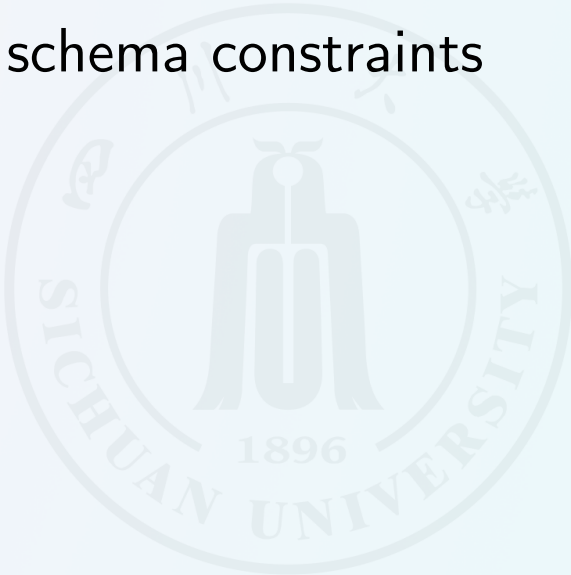
A system that is partition-tolerant can sustain any amount of network failure that doesn't result in a failure of the entire network. Data records are sufficiently replicated across combinations of nodes and networks to keep the system up through intermittent outages. **Partition Tolerance is not an option. It's a necessity**.

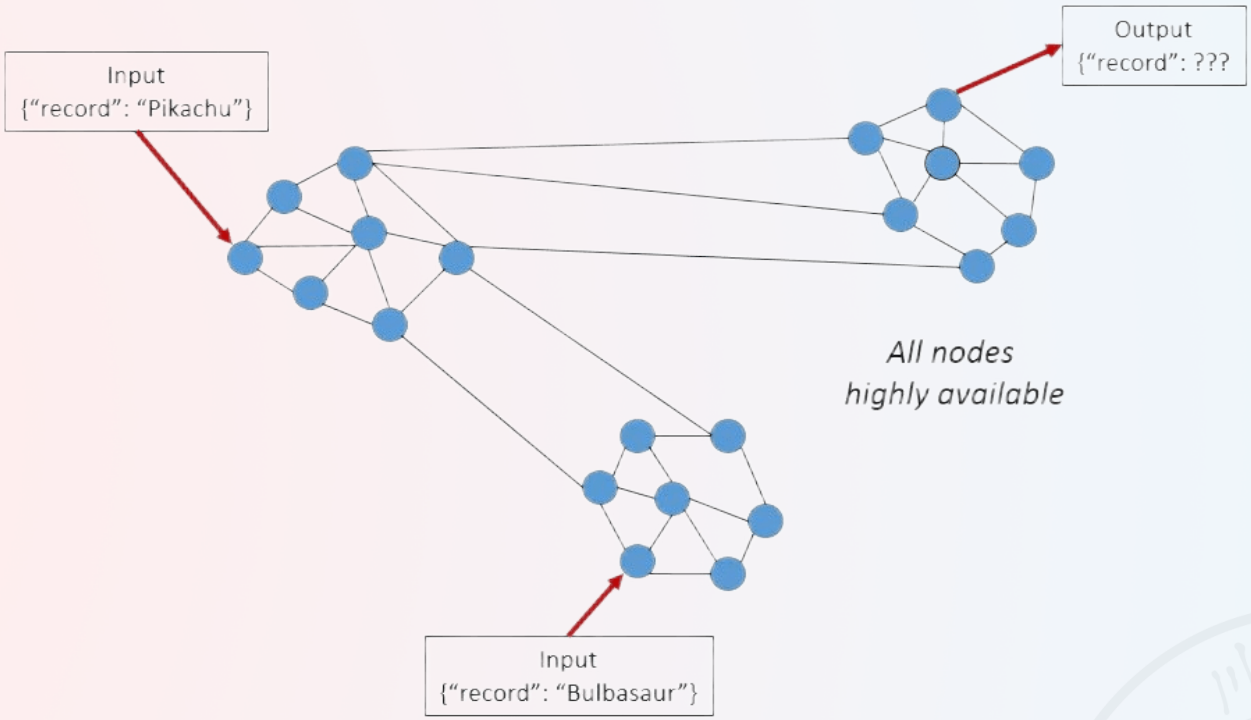


A system has consistency if a transaction starts with the system in a consistent state, and ends with the system in a consistent state. In this model, a system can shift into an inconsistent state during a transaction, but the entire transaction gets rolled back if there is an error during any stage in the process.

C in CAP != C in ACID

- They are different!
- CAP's C(onsistency) = sequential consistency
  - Similar to ACID's A(tomicity) = Visibility to all future operations
- ACID's C(onsistency) = Does the data satisfy schema constraints



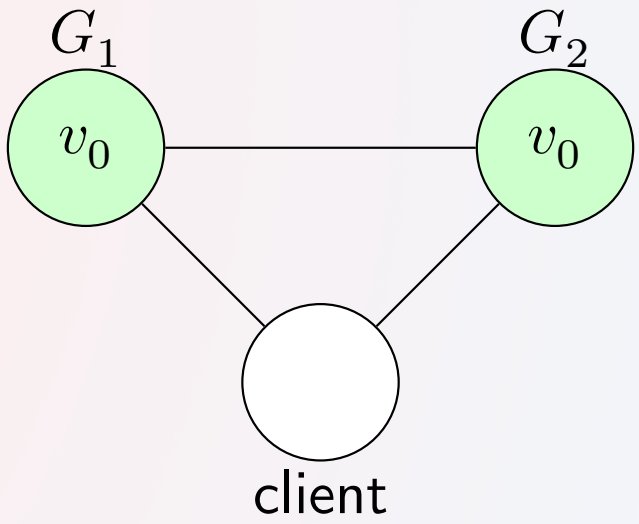


This condition states that every request gets a response on success/failure. Achieving availability in a distributed system requires that the system remains operational 100% of the time. Every client gets a response, regardless of the state of any individual node in the system.

Simple Proof

例 10

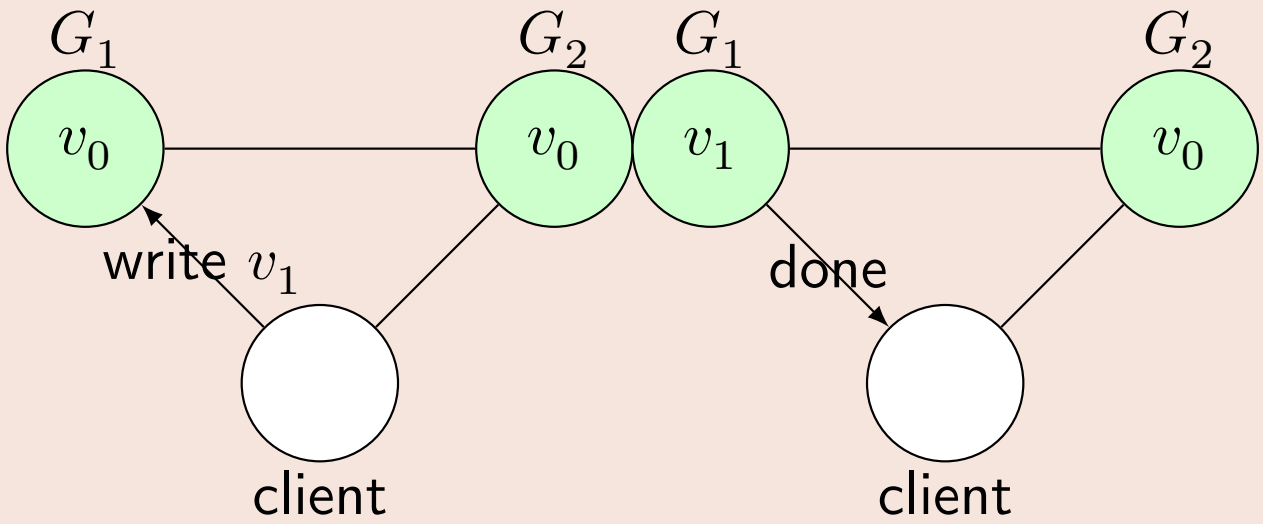
G_1 and G_2. Both of these servers are keeping track of the same variable, v, whose value is initially v_0.



Gilbert, Seth, and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." Acm Sigact News 33, no. 2 (2002): 51-59.

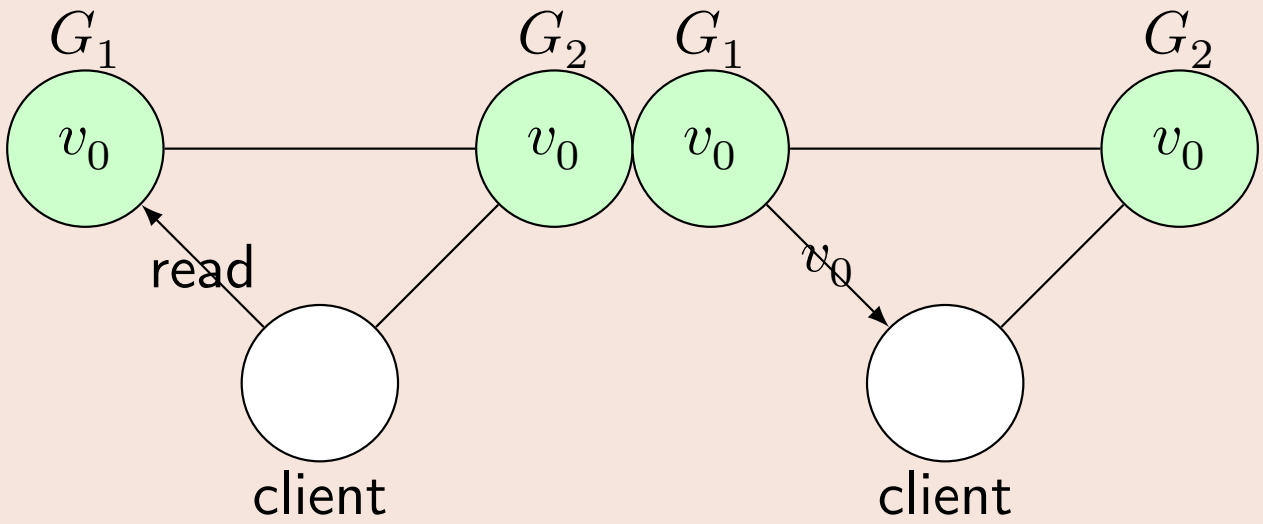
Simple Proof

write



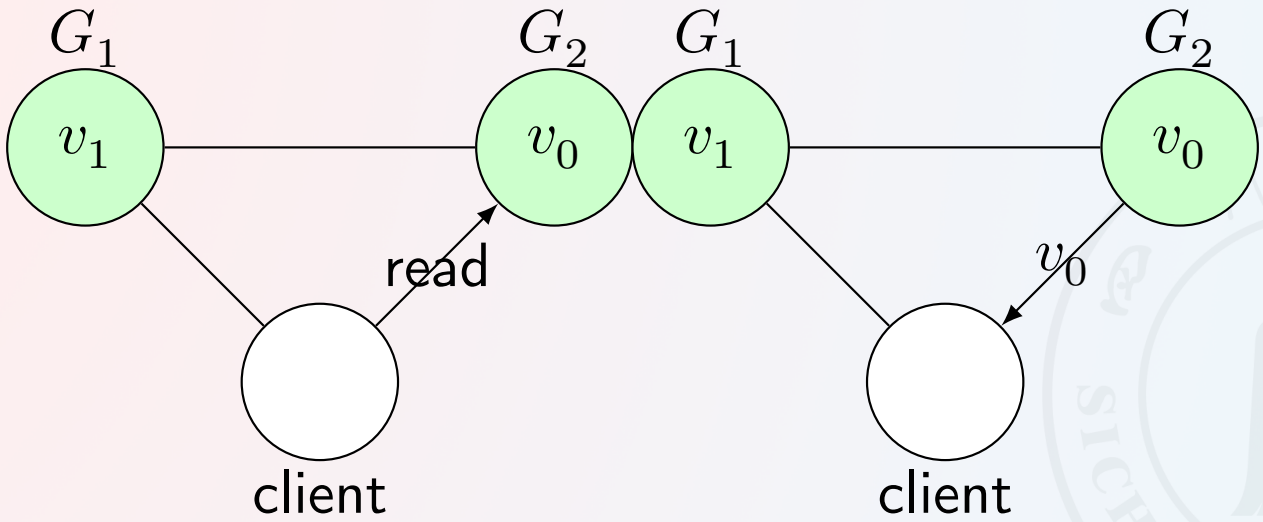
Simple Proof

read

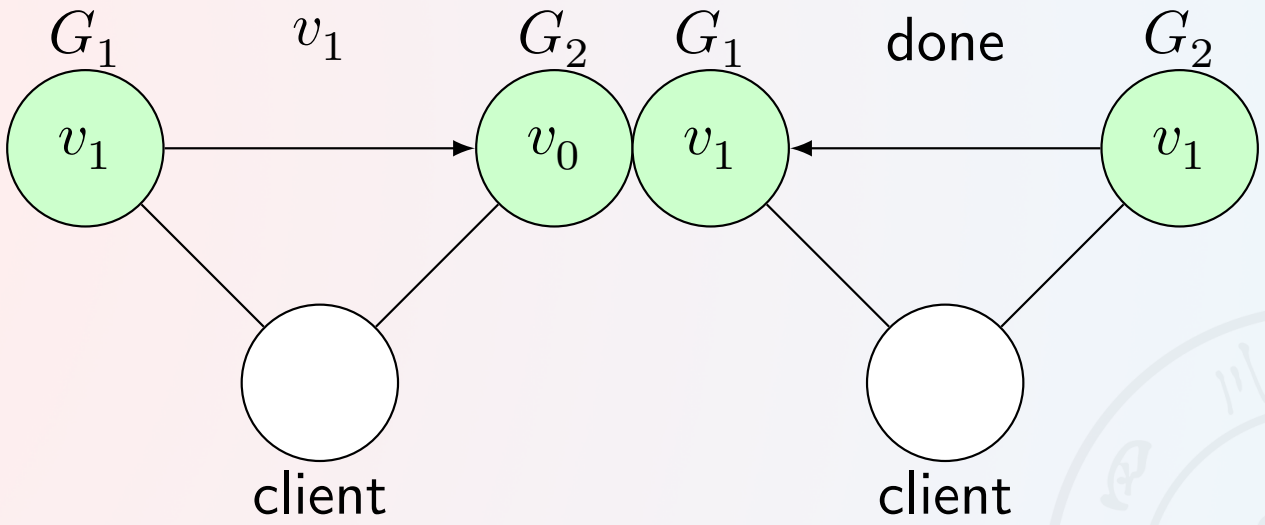


How Gilbert and Lynch describe consistency.
any read operation that begins after a write operation completes must return that value, or the result of a later write operation

Our client writes v_1 to G_1 and G_1 acknowledges, but when it reads from G_2, it gets stale data: v_0.

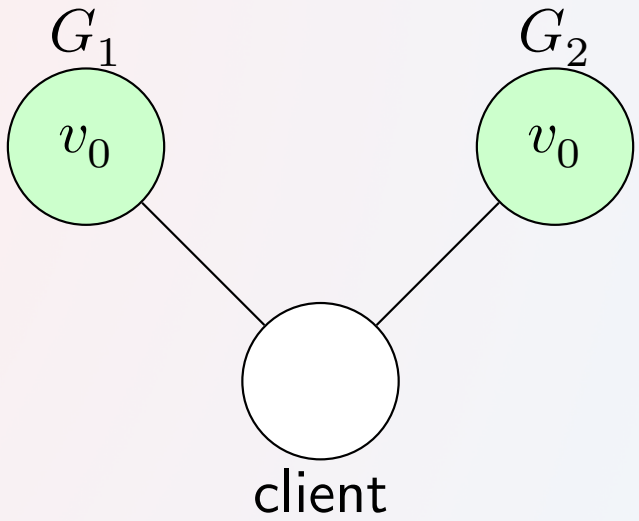


In **consistent system**, G_1 replicates its value to G_2 before sending an acknowledgement to the client.



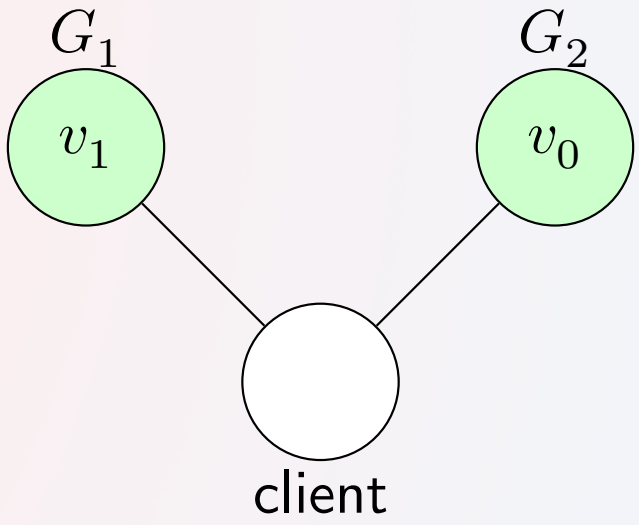
how Gilbert and Lynch describe availability.
every request received by a non-failing node in the system must result in a response

how Gilbert and Lynch describe partitions.
the network will be allowed to lose arbitrarily many messages sent from one node to another

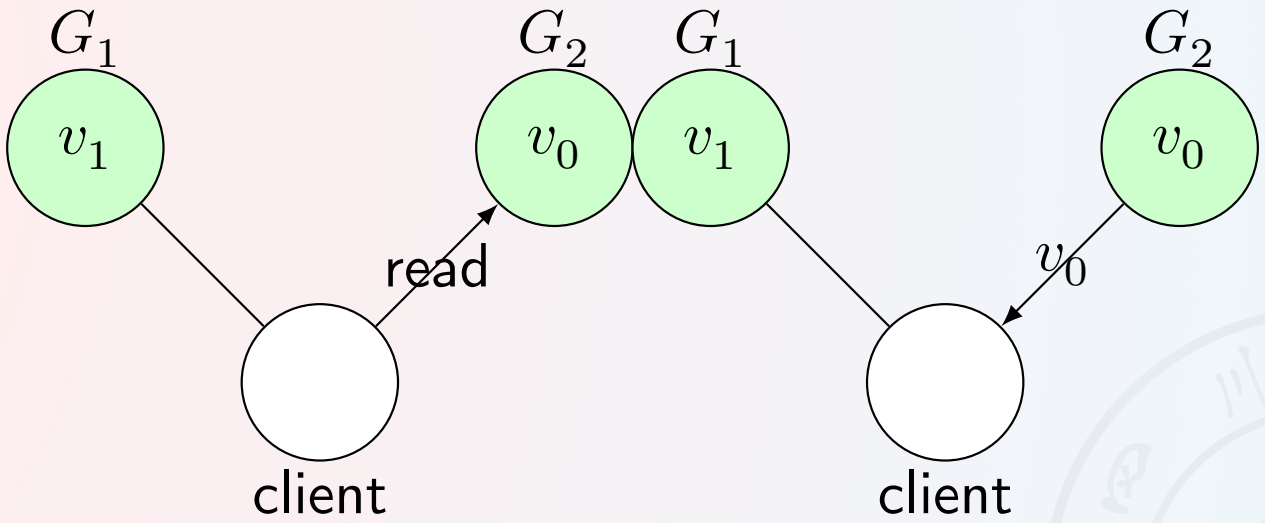


Our system has to be able to function correctly at this condition.

We have our client request that v_1 be written to G_1. Since our system is available, G_1 must respond. Since the network is partitioned, however, G_1 cannot replicate its data to G_2. Gilbert and Lynch call this phase of execution α_1.



Next, we have our client issue a read request to G_2. Again, since our system is available, G_2 must respond. And since the network is partitioned, G_2 cannot update its value from G_1. It returns v_0. Gilbert and Lynch call this phase of execution α_2.

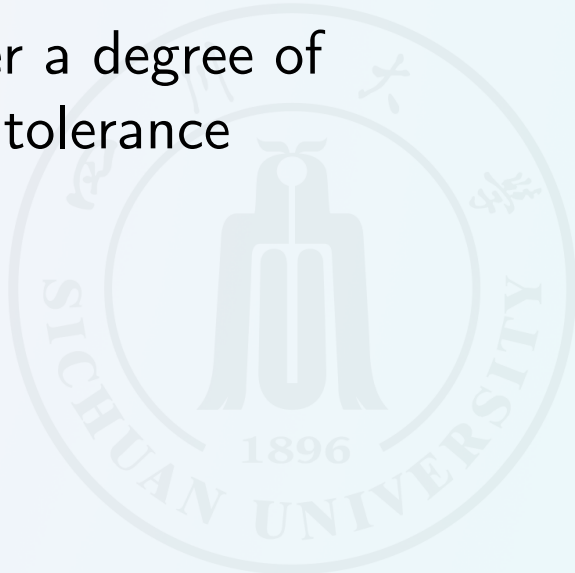


We assumed a consistent, available, partition tolerant system existed, but we just showed that there exists an execution for any such system in which the system acts inconsistently.

- "The "2 of 3" formulation was always misleading because it tended to **oversimplify** the tensions among properties. ...
- **CAP prohibits only a tiny part of the design space**: perfect availability and consistency in the presence of partitions, which are rare."

Eric Brewer: father of CAP

- Consistency and Availability is not a "binary" decision
- AP systems relax consistency in favor of availability –but are not inconsistent
- CP systems sacrifice availability for consistency but are not unavailable
- This suggests both AP and CP systems can offer a degree of consistency, and availability, as well as partition tolerance



The BASE model

Basically Available

Data is available most of the time. For instance, the data in the Beijing server won't be **immediately available** to the server in London. Also, database servers can and will fail, leading to downtime in availability.

Soft State

it may update even when it hasn't been written to. When the server in London finally gets data from Beijing, it will update —but not because data was written directly to it. In order for data to become available and consistent, it may be **written to the database at different times**.

Eventually Consistent

Eventually, the data in London will match the data in Beijing.

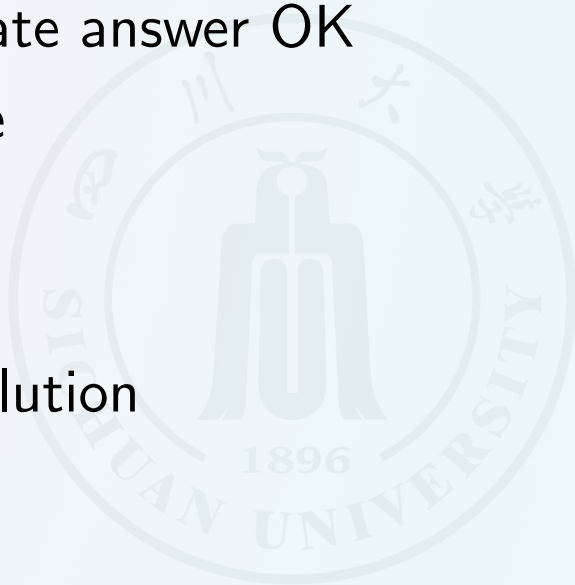
ACID (酸基) vs BASE (盐基)

ACID

- Strong consistency
- Isolation
- Focus on 'commit'
- Nested transactions
- Availability?
- Conservative
- Difficult evolution (e.g schema)

BASE

- Weak consistency
  - State data OK
- Availability first
- Best effort
- Approximate answer OK
- Aggressive
- Simpler
- Faster
- Easier evolution



例 11

If an ATM is disconnected from the network and when the partition eventually heals, the ATM sends sends a list of operations to the bank and the end balance will still be correct. The issue is obviously you might withdraw more money than you have so the end result might be consistent, but negative, which can't be compensated for by asking for the money back, so instead, the bank will reward you with an overdraft penalty.

The hidden philosophy is that you are trying to **bound and manage your risk**, yet still have all operations available.

Eric Brewer On Why Banks Are BASE Not ACID - Availability Is Revenue

Thanks
End of Chapter 21