CS-310 Scalable Software Architectures

Lecture 13:
Distributed NoSQL Databases

Steve Tarzia

Last Time: Push Notifications

- Traditional web/app design uses a **client-server** model, but sometimes we want to **push** data to client instead of client always **pulling**.
- Asynchronously sending data to clients can be a challenge.
- Mobile OSes have special **push notification services**.
  - Allows a single connection to be shared by all apps on the phone.
  - Allows notifications to be delivered even if app is not running.
- Web browsers can use **Websockets** or **Long-polling.**
  - In both cases, client is connected to one machine and service must somehow relay messages to that connection.

Recall from Lecture 9: Scaling SQL Databases

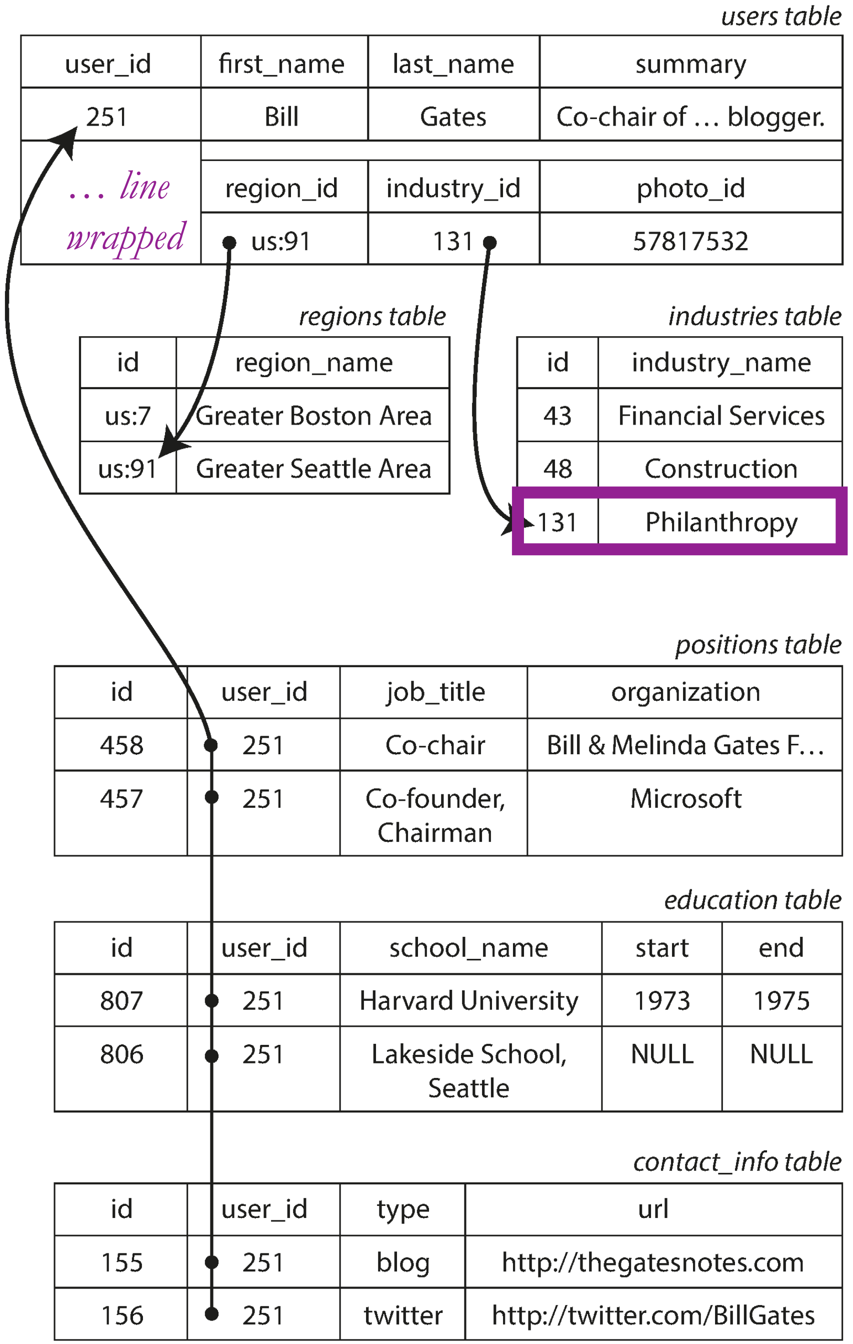
- **Read replicas** horizontally scale databases for reading.
  - Writes are done in one place and propagated to many replicas.
  - Data on a given replica may lag behind master, but it's self-**consistent.**
  - Works well if writes are much less common than reads.
- Horizontal scaling of writes suggests **data partitioning**.
  - Each data row/element is assigned a single "home" (or a constant number).
  - If not, each node must accept all writes, which is not scalable.
- **Sharding** is data partitioning for SQL/relational DBs.
  - Works well for queries that can be handled within a single shard.
  - Sharding divides data along just one dimension, so inevitably some queries will involve all the nodes, and thus will not be scalable.

Review of **Sharding:**

- **Splits data** among many machines.
- Accept **writes** on all machines.
- But the data partitioning is done manually.
  - Programmer chooses a sharding key or rule, and to write code that joins results from the different shards.
- Works well for queries that can be handled within a single shard.
- If we keep the relational model, with **normalized** data, many queries will involve all the nodes, so scaling is limited.
- NoSQL databases all solve this problem by **denormalizing** data, meaning that data is duplicated to isolate queries to one node.

Normalized data

- A **normalized** relational database has no duplication of data.
- References (foreign keys) point to shared data.
- Eg., at right, the Philanthropy industry is shared by many LinkedIn users.
    - In effect, many users are related to each other by all being linked to that industry
- To optimally partition the rows into shards, we could solve a balanced graph partitioning problem.



users table

… line wrapped

user_id	first_name	last_name	summary	region_id	industry_id	photo_id
251	Bill	Gates	Co-chair of … blogger.	us:91	131	57817532

regions table

id	region_name
us:7	Greater Boston Area
us:91	Greater Seattle Area

industries table

id	industry_name
43	Financial Services
48	Construction
131	Philanthropy

positions table

id	user_id	job_title	organization
458	251	Co-chair	Bill & Melinda Gates F…
457	251	Co-founder, Chairman	Microsoft

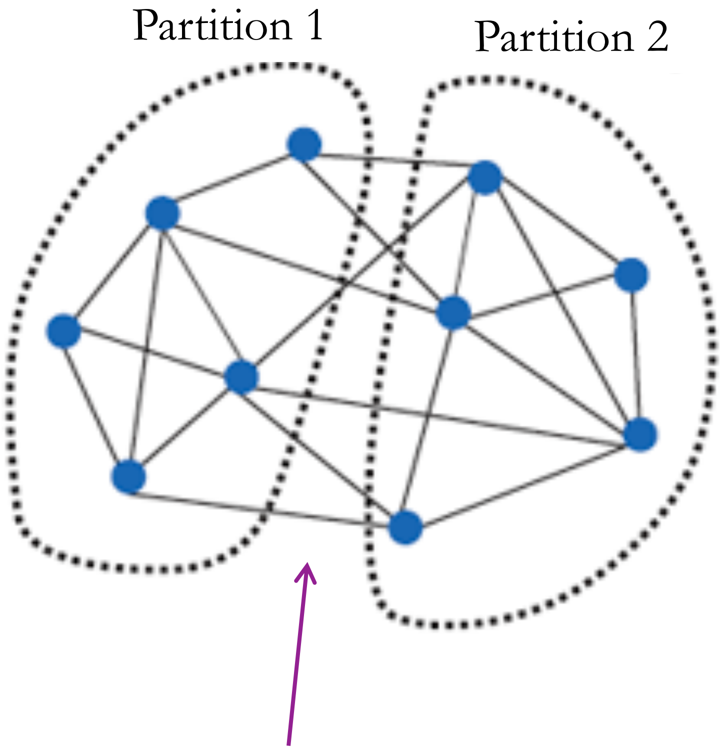
education table

id	user_id	school_name	start	end
807	251	Harvard University	1973	1975
806	251	Lakeside School, Seattle	NULL	NULL

contact_info table

id	user_id	type	url
155	251	blog	http://thegatesnotes.com
156	251	twitter	http://twitter.com/BillGates

Graph partitioning model for DB sharding



Edges between partitions imply data transfers between nodes.

- Nodes represent database rows.
- Edges represent references (foreign keys)

Task: assign the rows to shards (partition the nodes),

Such that:

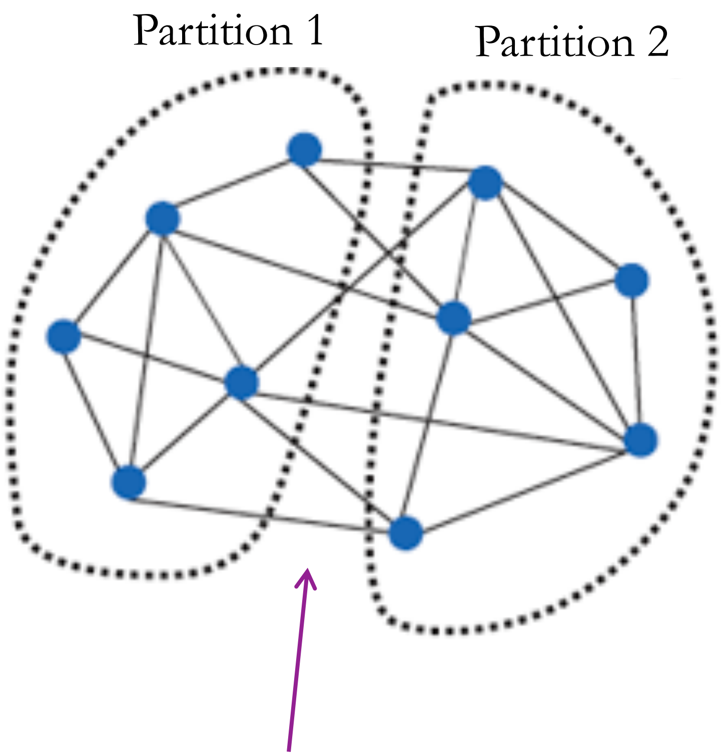
- Total edges between partitions is minimized.

  (Need to fetch data from another shard for a JOIN is minimized.)
- Nodes per partition is roughly balanced.

  (Data stored on each shard is balanced.)

Reference: Facebook's social graph sharding: https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

Partitioning challenges



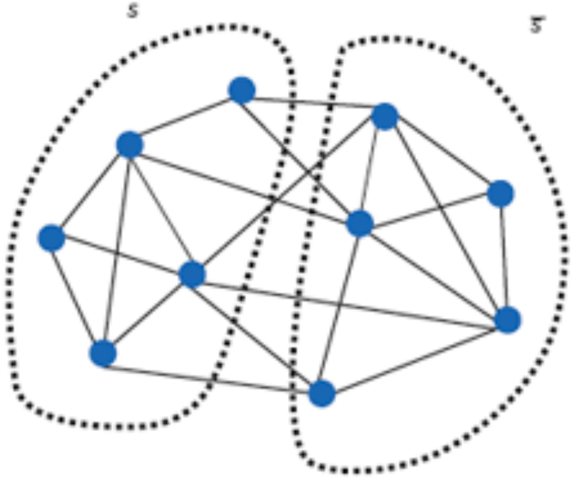
Edges between partitions imply data transfers between nodes.

- Solving this problem is NP-complete.
  (But we can approximately solve it fairly well.)
- This model's cost function is too simplistic:
  - Some rows are fetched more often.
  - An edge can pull data transitively from lots of nodes.
    Ie., the cost of a reference can vary dramatically.
- Even with an optimal partitioning, we still have data references between partitions.
- How does the data (graph structure) affect the solution quality?
  - Random interconnections hurt.
  - Nodes with high degree (many edges) hurt.
  - Structured, independent relationships are easy.
  - Nodes with one edge (spurs) are easy.

STOP and THINK

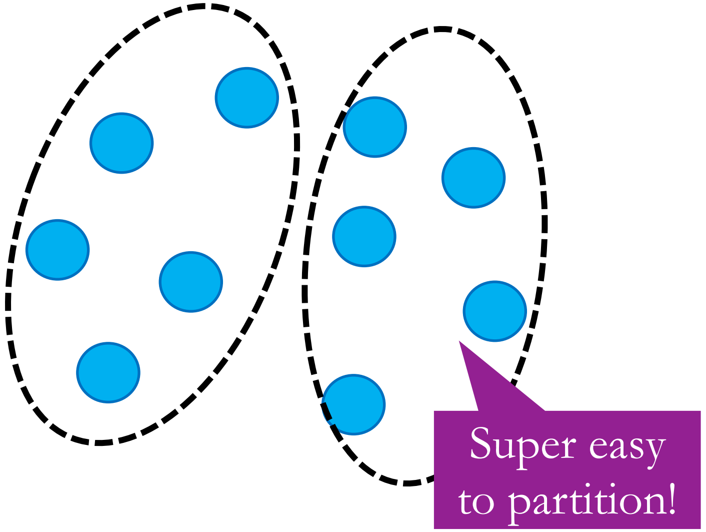
The jump from SQL to NoSQL

SQL sharding



- Eliminating the edges (references) would make the data partitioning problem trivial!
- Foreign keys (references) and JOINs (dereferences) are fundamental to SQL and relational databases.

NoSQL partitioning

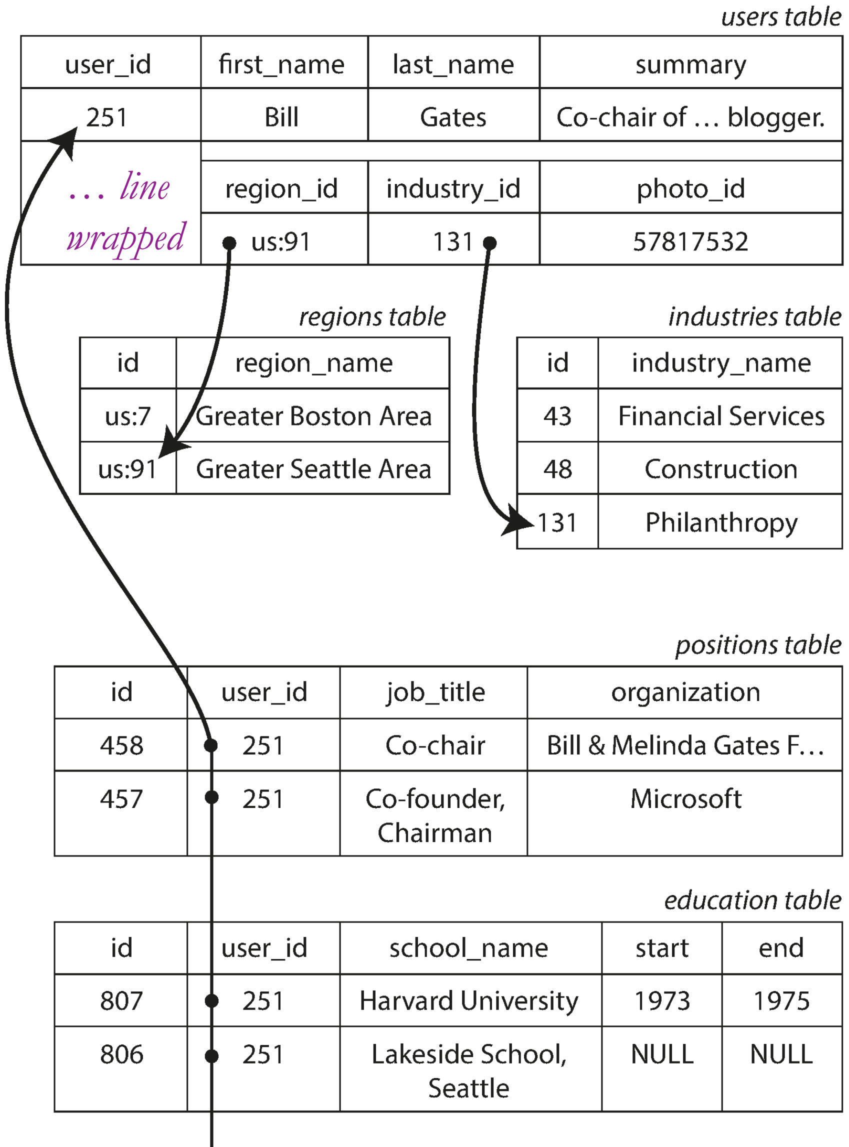


Super easy to partition!

- Removing the ability to create references gives us a **NoSQL** database.
- Instead of following references with JOINs, we store **denormalized** data, with copies of referenced data.

From Normalized … to Denormalized Data

SQL



users table

user_id	first_name	last_name	summary
251	Bill	Gates	Co-chair of … blogger.

… line wrapped

region_id	industry_id	photo_id
us:91	131	57817532

regions table

id	region_name
us:7	Greater Boston Area
us:91	Greater Seattle Area

industries table

id	industry_name
43	Financial Services
48	Construction
131	Philanthropy

positions table

id	user_id	job_title	organization
458	251	Co-chair	Bill & Melinda Gates F…
457	251	Co-founder, Chairman	Microsoft

education table

id	user_id	school_name	start	end
807	251	Harvard University	1973	1975
806	251	Lakeside School, Seattle	NULL	NULL

NoSQL

user_id	value
251	

```
{
  "first_name": "Bill",
  "last_name:" "Gates",
  "summary": "Co-chair of … blogger.",
  "region": "Greater Seattle Area",
  "industry": "Philanthropy",
  "photo_id": 57817532,
  "positions": [
    { "job_title": "Co-chair",
      "organization": "Bill & Melinda …" },
    { "job_title": "Co-founder, Chairman",
      "organization": "Microsoft" }
  ],
  "education": [
    { "school_name": "Harvard University",
      "start": 1973,
       "end": 1975 }
    { "school_name": "Lakeside School, …"}
  ]
}
```

NoSQL rationale

key	value

user251

```
{
   "first_name": "Bill",
   "last_name": "Gates",
   "summary": "Co-chair of … blogger.",
   "region": "Greater Seattle Area",
   "industry": "Philanthropy",
   "photo_id": 57817532,
   "positions": [
      { "job_title": "Co-chair",
        "organization": "Bill & Melinda …" },
      { "job_title": "Co-founder, Chairman",
        "organization": "Microsoft" }
   ],
   "education": [
      { "school_name": "Harvard University",
        "start": 1973,
         "end": 1975 }
      { "school_name": "Lakeside School, …"}
   ]
}
```

user444

```
{
  first_name: "Steve",
```

One **key** is indexed.

Precise format of **value** varies by NoSQL DB type.

Why just one column?

- Without references, it's impossible to define finite/fixed columns (a schema).
- Consider "positions": we don't know how many position columns to add.
- Some NoSQL DBs allow multiple columns, but each row can have different columns ("wide columns")

Why just one table?

- Some NoSQL DBs allow multiple tables, but since rows can have any format, it's kind of meaningless.

NoSQL DBs are **key-value stores.**

Hashing is the basis of distributed NoSQL DBs

- A ***hash*** is an algorithm that takes a value and returns a pseudo-random value derived from it.
- It’s a *constant* but *unpredictable* mapping
  - A long sequence of arithmetic operations
- MD5 is a standard hash function:
  - `“Steve”` → `f6e997429bf8cb7b3b98b310a9f7ca30`
  - `“steve”` → `2666b87c682f5072f62bab0955d485ce`
  - `“Janice”` → `3837607db4754c036425cb1b2a7c8766`
  - `“1”` → `b026324c6904b2a9cb4b88d6d61c81d1`
  - `“Steve”` → `f6e997429bf8cb7b3b98b310a9f7ca30`
  - `tale_of_two_cities.txt` *(806,878 characters)* → `3ab56b74562a714a5638f94446581977`
- The same input always gives the same output
- Length of the input can vary, but output has fixed length

This hash is case sensitive

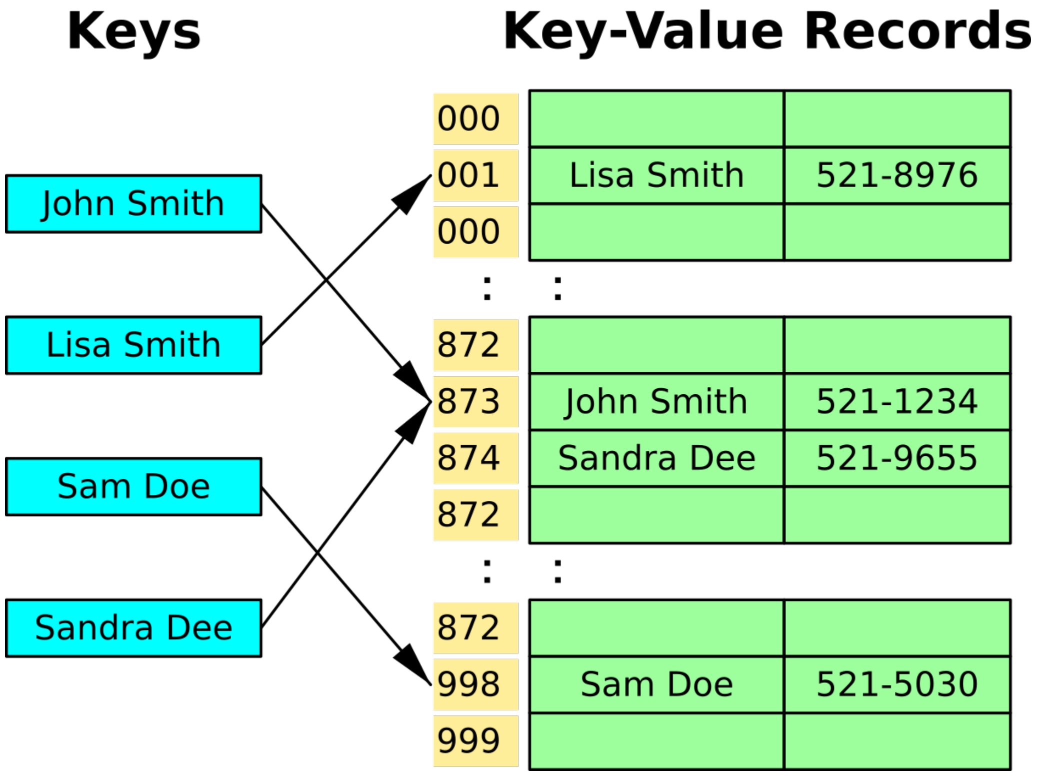
Hash Table

- Stores (***key***, ***value***) pairs
  - This abstract data type is called a ***dictionary***, or ***map***.
- For example:
  - A word and its definition.
    - “word” → “a single distinct meaningful element of speech or writing, …”
    - “hash” → “a dish of cooked meat cut into small pieces and cooked again, …”
  - A database table’s primary key and the rest of the columns in the row:
    - StaffID → [StfFirstName, StfLastName, StfStreetAddress, StfCity, StfState, ...]
    - 98005 → [“Suzanne”, “Viescas”, “15127 NE 24th, #383”, …]
    - 98007 → [“Gary”, “Hallmark”, “Route 2, Box 203B”, …]

key ***value***

Hash Table mechanics

- ***Hash the key*** to determine the address where the value is stored

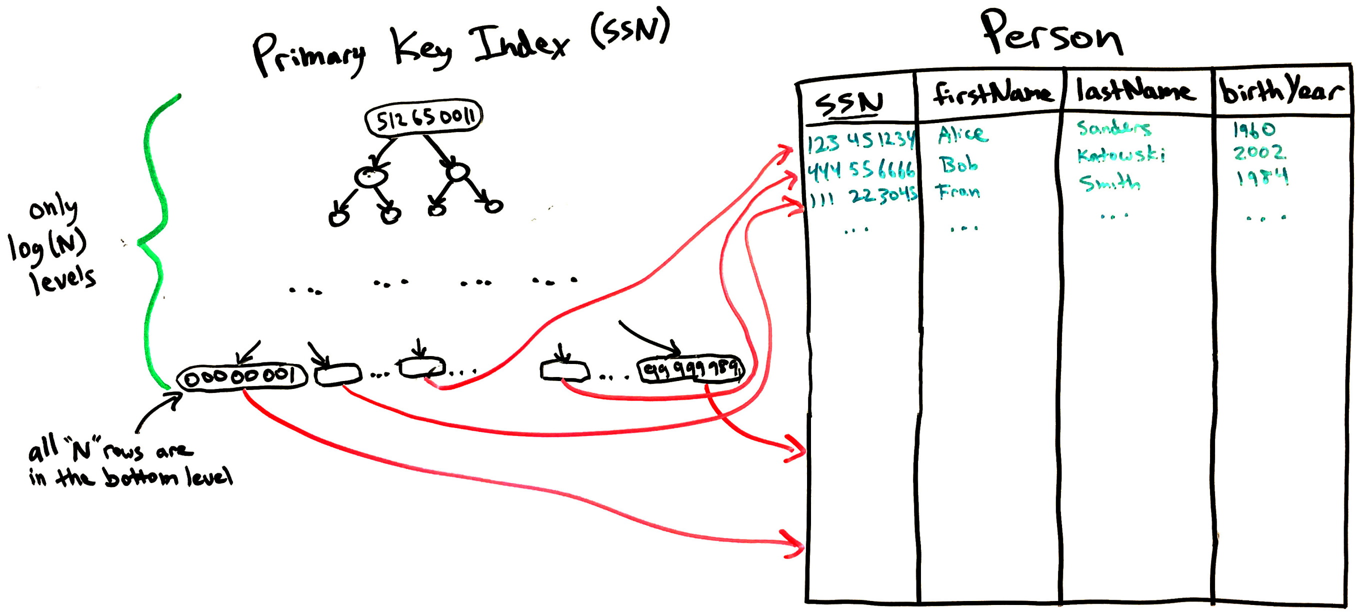


- If the address is already filled, then use the next open slot
  - This is called a *collision* and there are other strategies besides “linear probing”

Hash Indexes in SQL databases

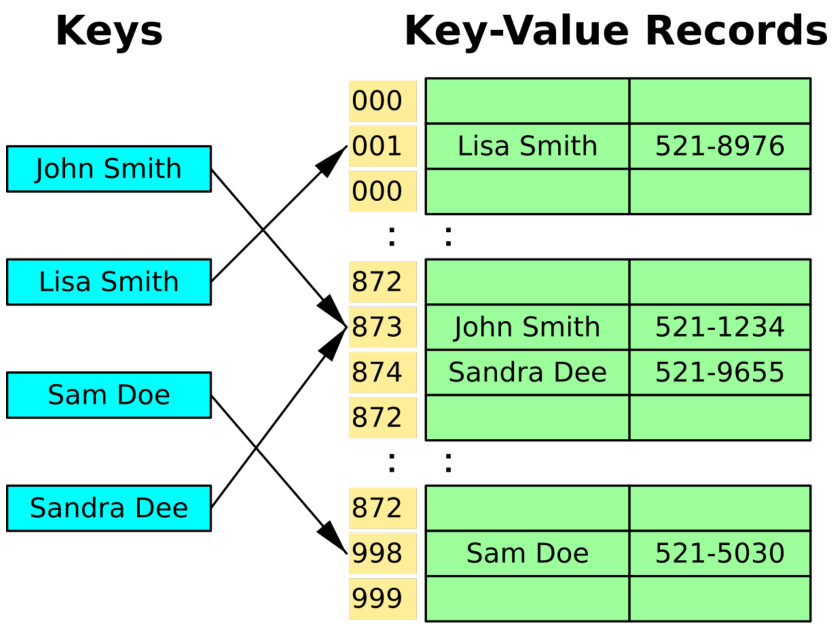
- A hash table is an alternative to a search tree
    - It lets you find the data in one step!
    - However, it does not support efficient range queries.
        - A hash table scatters data randomly, so walking though a range is difficult.

Tree-based table index



SSN	firstName	lastName	birthYear
123 45 1234	Alice	Sanders	1960
444 55 6666	Bob	Katowski	2002
111 22 3045	Fran	Smith	1984
...	...	...	...

Hash-based table index

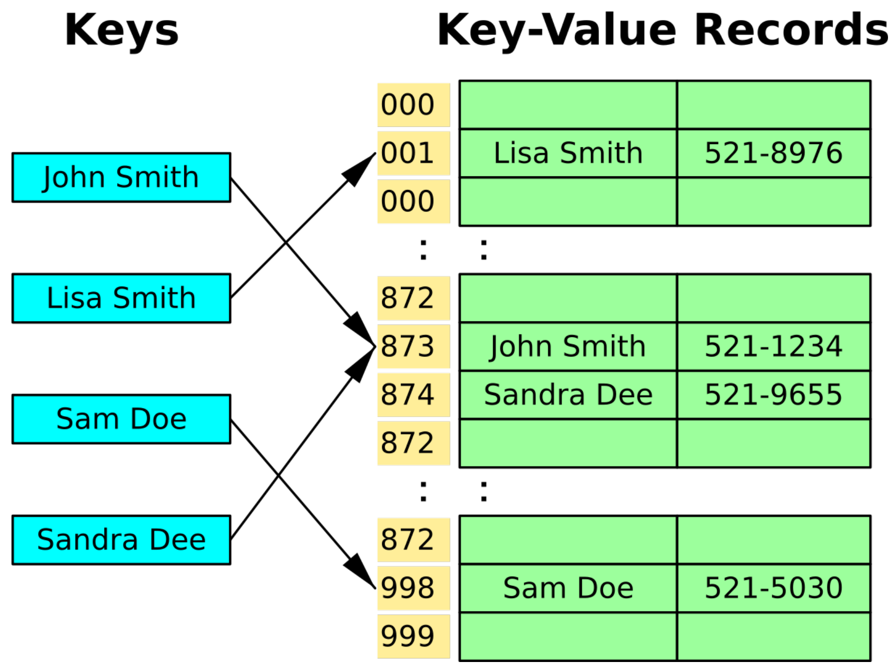


000		
001	Lisa Smith	521-8976
000		
:	:	
872		
873	John Smith	521-1234
874	Sandra Dee	521-9655
872		
:	:	
872		
998	Sam Doe	521-5030
999		

Distributed Hash Table

- Each cluster node is responsible for a range of hash values
- Each client gets the list of nodes and the range assigned to each.
- When querying for a key's value, client computes the key hash to determine which node to query for the data:

Hash of the key **partitions** the data



Node 1 stores hash values 000-332

Node 2 stores hash values 333-665

Node 3 stores hash values 666-999

DHT is a NoSQL database

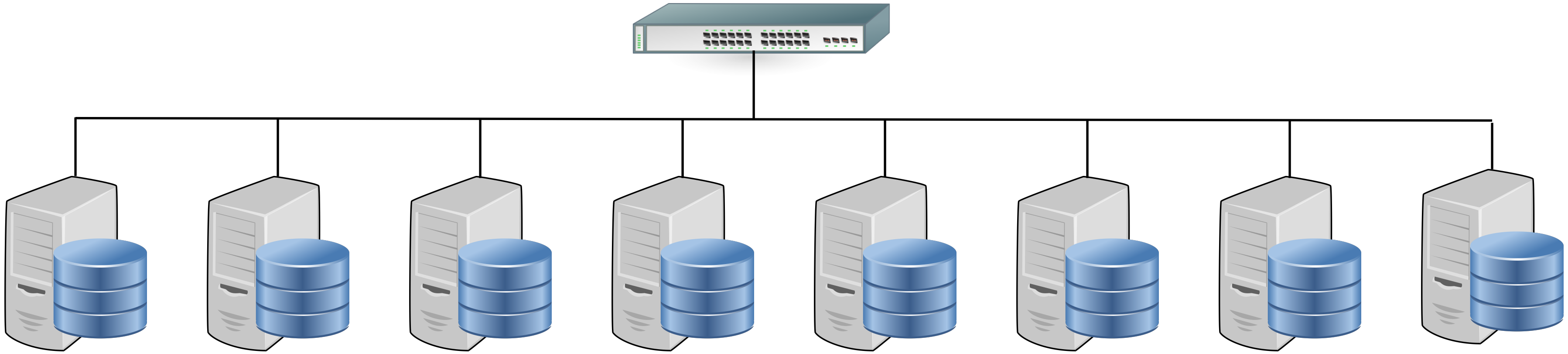
- NoSQL databases are distributed key-value stores
- Like one big table with just a primary key
- They have a map/dictionary interface, do not support SQL queries.
    - You can only:
        - ***get*** a value for a key.
        - ***put*** a value for a key.
    - Each operation only affects the node(s) storing that key
        - Very **scalable**! (can grow large without slowing down)
- If we wanted to support full SQL, JOINs would have to pull data from many nodes in the cluster and performance would be slow.

What limits the scalability?

STOP and THINK

Distributed, **shared-nothing** architecture

- Create a **cluster** of computers connected to each other.
- Each **node** in the cluster stores a fraction of the data set.



- Distributed database examples:
  - MongoDB, Cassandra, Amazon DynamoDB, …
- Distributed filesystems also use the same basic idea:
  - Hadoop HDFS, Google File System (Colossus, BigTable), Amazon S3, …

NoSQL downsides :(

key	value

user251

```
{
   "first_name": "Bill",
   "last_name:" "Gates",
   "summary": "Co-chair of … blogger.",
   "region": "Greater Seattle Area",
   "industry": "Philanthropy",
   "photo_id": 57817532,
   "positions": [
      { "job_title": "Co-chair",
        "organization": "Bill & Melinda …" },
      { "job_title": "Co-founder, Chairman",
        "organization": "Microsoft" }
   ],
   "education": [
      { "school_name": "Harvard University",
        "start": 1973,
         "end": 1975 }
      { "school_name": "Lakeside School, …"}
   ]
}
```

user444

```
{
   "first_name": "Steve",
```

- Just one indexed column (the key).
  - Because index is built with hash-based partitioning.
- Denormalized data is duplicated.
  - Wastes space.
  - Cannot be edited in one place.
  - Eg., "Greater Seattle Area" is repeated in many user profiles instead of "region:91"
- References are possible, but:
  - Following the reference requires another query, probably to another node.
  - There is no constraint checking (refs can become invalid after delete).

Normalization thought experiment

key	value
user251	{ "first_name": "Bill", "last_name": "Gates", "summary": "Co-chair of … blogger.", "region": "us:91" "industry": 131 "photo_id": 57817532, "positions": [458, 457], "education": [807, 806] }
reg:us:91	{ "region_name": "Greater Seattle Area" }
ind:131	{ "industry_name": "Pilanthropy"
pos:458	{ "user_id": 251, "job_title": "Co-chair", "organization": "Bill and Melinda Gates …" }
pos:457	{ "user_id": 251, "job_title": "Co-founder, Chairman", "organization": "Microsoft"
edu:807	{ "user_id": 251, "school_name": "Harvard University", "start": 1973, "end": 1975 }

- What happens if we try to store normalized data, like this, in a NoSQL database?
- Is it possible?
- Why isn't it done?

- It's possible, but you would need many serial queries to many different DB nodes to fetch the user's profile.
- References are not enforced by the schema, so they can become broken.

Summary

- **Data partitioning** is necessary to divide write load among nodes.
    - Should minimize references between partitions.
    - Can be treated as a graph partitioning problem.
    - SQL sharding was a special case of data partitioning, done in app code.
- **NoSQL** databases make partitioning easy by eliminating references.
- Without references, data becomes **denormalized**.
    - Duplicated data consumes more space, can become inconsistent.
- **NoSQL databases** are very scalable, but they provide only a very simple **key-value** abstraction. One key is indexed.
- **Distributed Hash Table** can implement a NoSQL database.
    - The hash space is divided evenly between storage nodes.
    - Client computes hash of key to determine which node should store data.