Executing Queries on a Sharded Database

Neha Narula

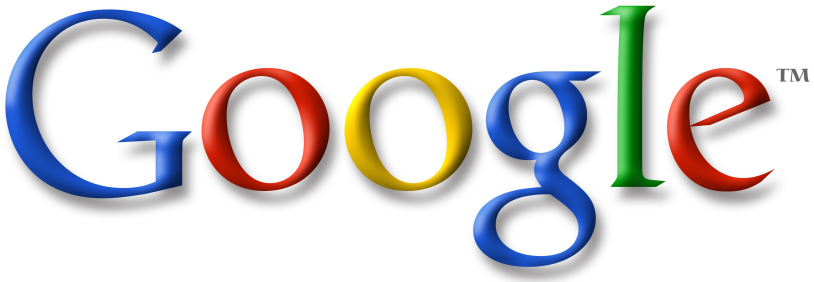
September 25, 2012

Choosing and Scaling Your Datastore

Neha Narula

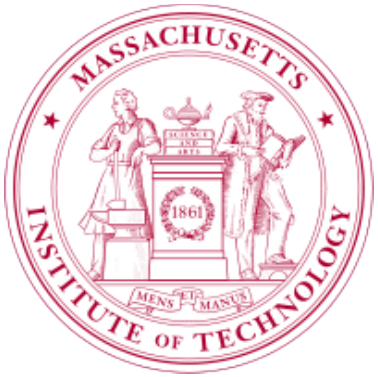
September 25, 2012

Who Am I?



Froogle
Blobstore
Native Client

NEWS.ME



In This Talk

- What to think about when choosing a datastore for a web application
- Myths and legends
- Executing distributed queries
- My research: a consistent cache

Every so often…

Friends ask me for advice when they are building a new application

Friend

“Hi Neha! I am making a new application.

I have heard MySQL sucks and I should use NoSQL.

I am going to be iterating on my app a lot

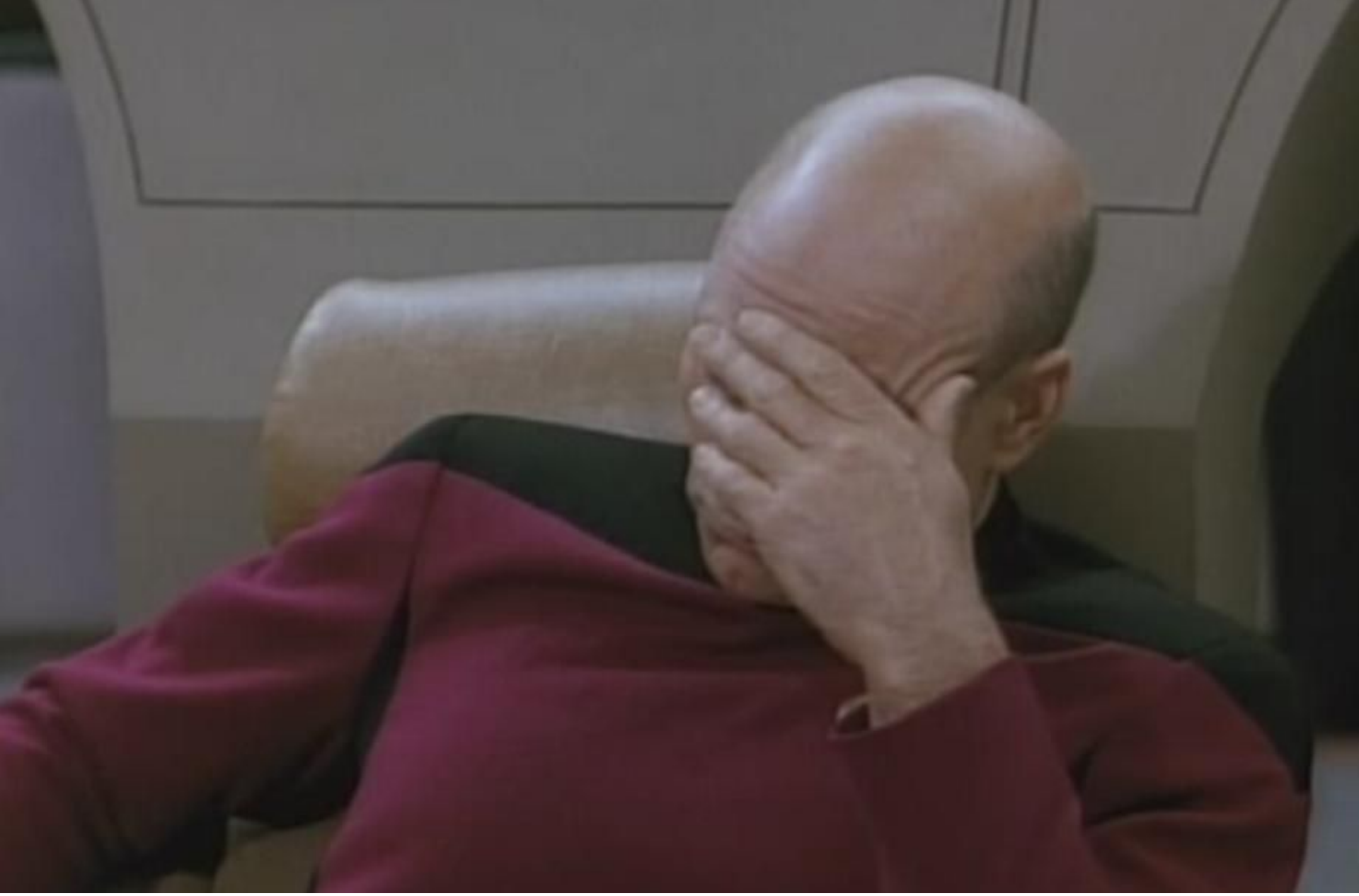
I don't have any customers yet

and my current data set could fit on a thumb drive from 2004, but…

Can you tell me which NoSQL database I should use?

And how to shard it?"

Neha



http://knowyourmeme.com/memes/facepalm


PostgreSQL
Project Voldemort
A distributed database.
amazon
SimpleDB
Tokyo
Cabinet
8192PiB
membase
MySQL
memcached
redis
APACHE
HBASE
riak
Cassandra
memsql

What to Think about When You're Choosing a Datastore

Hint: Not Scaling

Development Cycle

- Prototyping
  - Ease of use
- Developing
  - Flexibility, multiple developers
- Running a real production system
  - Reliability
- SUCCESS! Sharding and specialized datastorage
  - We should only be so lucky

Getting Started

First five minutes – what's it like?

Idea credit: Adam Marcus, The NoSQL Ecosystem, HPTS 2011

via Justin Sheehy from Basho

First Five Minutes: Redis

Here's a 30 second guide to getting started with Redis:

```
$ wget http://redis.googlecode.com/files/redis-1.01.tar
$ tar -xzf redis-1.01.tar.gz
$ cd redis-1.01
$ make
$ ./redis-server
```

And that's it—you now have a Redis server running on port 6379. No need even for a `./configure` or `make install`. You can run `./redis-benchmark` in that directory to exercise it a bit.

Let's try it out from Python. In a separate terminal:

```
$ cd redis-1.01/client-libraries/python/
$ python
>>> import redis
>>> r = redis.Redis()
>>> r.info()
{u'total_connections_received': 1, ... }
>>> r.keys('*') # Show all keys in the database
[]
>>> r.set('key-1', 'Value 1')
'OK'
>>> r.keys('*')
[u'key-1']
>>> r.get('key-1')
u'Value 1'
```

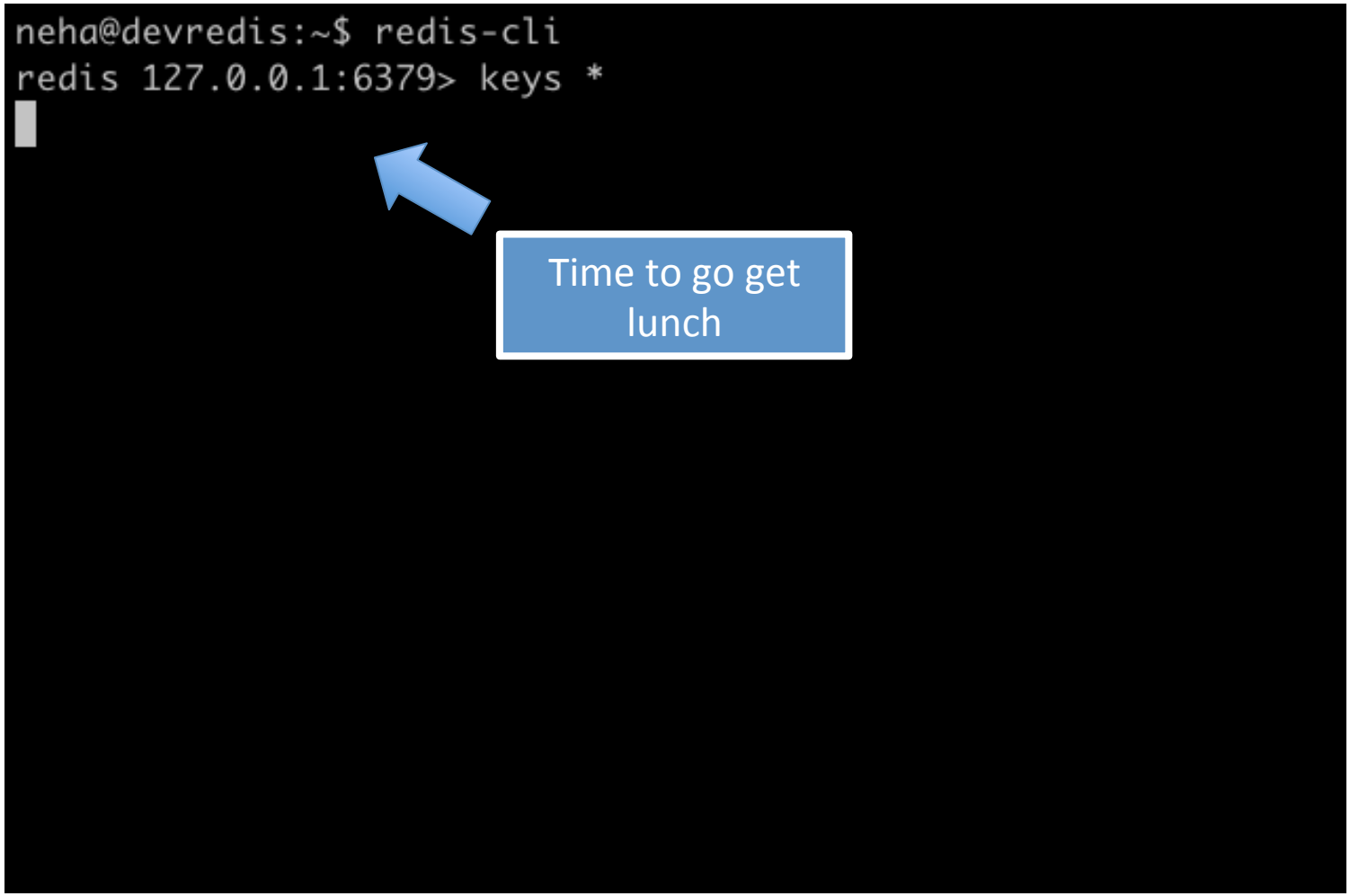
http://simonwillison.net/2009/Oct/22/redis/

First Five Minutes: MySQL

Developing

- Multiple people working on the same code
- Testing new features => new access patterns
- New person comes along…

Redis


neha@devredis:~$ redis-cli
redis 127.0.0.1:6379> keys *
Time to go get lunch

MySQL

```
mysql> show tables;
+-----------------+
| Tables_in_cards |
+-----------------+
| cards           |
+-----------------+
1 row in set (0.00 sec)

mysql> describe cards;
+--------------+------------------+------+-----+---------+----------------+
| Field        | Type             | Null | Key | Default | Extra          |
+--------------+------------------+------+-----+---------+----------------+
| id           | int(11)          | NO   | PRI | NULL    | auto_increment |
| number       | blob             | YES  |     | NULL    |                |
| magic_number | varchar(50)      | YES  |     | NULL    |                |
| created      | int(10) unsigned | YES  |     | NULL    |                |
| request_id   | varchar(50)      | YES  |     | NULL    |                |
+--------------+------------------+------+-----+---------+----------------+
5 rows in set (0.01 sec)

mysql> select * from cards limit 3;
+----+------------+--------------+-----------+------------+
| id | number     | magic_number | created   | request_id |
+----+------------+--------------+-----------+------------+
|  1 | 1234567789 | lskdjfklsdjf | 123456789 | slkdjfkdl  |
|  2 | 1111111111 | sdlfkjdlksfj | 123456789 | slkdjfkdl  |
|  3 | 420878770  | ! G?)?       | 123456789 | sljdkfldsk |
+----+------------+--------------+-----------+------------+
3 rows in set (0.00 sec)

mysql>
```

Questions

- What is your mix of reads and writes?
- How much data do you have?
- Do you need transactions?
- What are your access patterns?
- What will grow and change?
- What do you already know?

Reads and Writes

- Write optimized vs. Read optimized
- MongoDB has a global write lock per process
- No concurrent writes!

Size of Data

- Does it fit in memory?
- Disk-based solutions will be slower
- Worst case, Redis needs 2X the memory of your data!
  - It forks with copy-on-write


```
sudo echo 1 > /proc/sys/vm/overcommit_memory
```

Requirements

- Performance
  - Latency tolerance
- Durability
  - Data loss tolerance
- Freshness
  - Staleness tolerance
- Uptime
  - Downtime tolerance

Performance

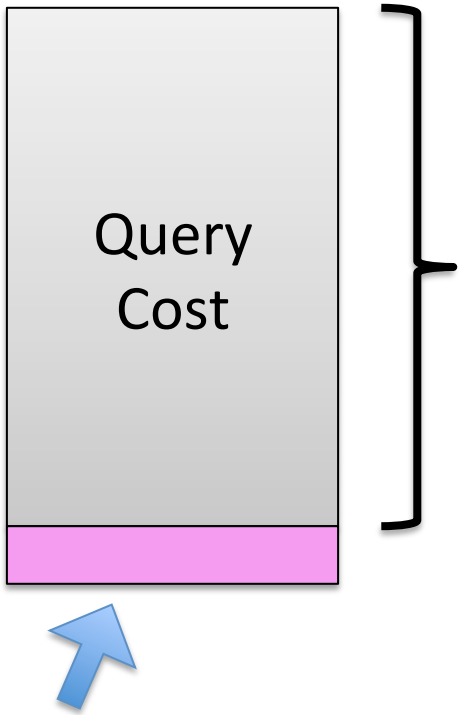
- Relational databases are considered slow
- But they are doing a lot of work!
- Sometimes you need that work, sometimes you don’t.

Simplest Scenario

- Responding to user requests in real time
- Frequently read data fits in memory
- High read rate
- No need to go to disk or scan lots of data

Datastore CPU is the bottleneck

Cost of Executing a Primary Key Lookup



Actually retrieving data

- Receiving message
- Instantiating thread
- Parsing query
- Optimizing
- Take out read locks
- (Sometimes) lookup in an index btree
- Responding to the client

Options to Make This Fast

- Query cache
- Prepared statements
- Handler Socket
  - 750K lookups/sec on a single server

Lesson: Primary key lookups can be fast no matter what datastore you use

Flexibility vs. Performance

- We might want to pay the overhead for query flexibility
- In a primary key datastore, we can only ask queries on primary key
- SQL gives us flexibility to change our queries

Durability

- Persistent datastores
  - Client: write
  - Server: flush to disk, then send “I completed your write”
  - CRASH
  - Recover: See the write
- By default, MongoDB does not fsync() before returning to client on write
  - Need j:true
- By default, MySQL uses MyISAM instead of InnoDB

Specialization

- You know your
  - query access patterns and traffic
  - consistency requirements
- Design specialized lookups
  - Transactional datastore for consistent data
  - Memcached for static, mostly unchanging content
  - Redis for a data processing pipeline
- Know what tradeoffs to make

Ways to Scale

- Reads
  - Cache
  - Replicate
  - Partition
- Writes
  - Partition data amongst multiple servers

Lots of Folklore

Myths of Sharded Datastores

- NoSQL scales better than a relational database
- You can't do a JOIN on a sharded datastore

MYTH: NoSQL scales better than a relational database

- Scaling isn't about the datastore, it's about the application
  - Examples: FriendFeed, Quora, Facebook
- Complex queries that go to all shards don't scale
- Simple queries that partition well and use only one shard do scale

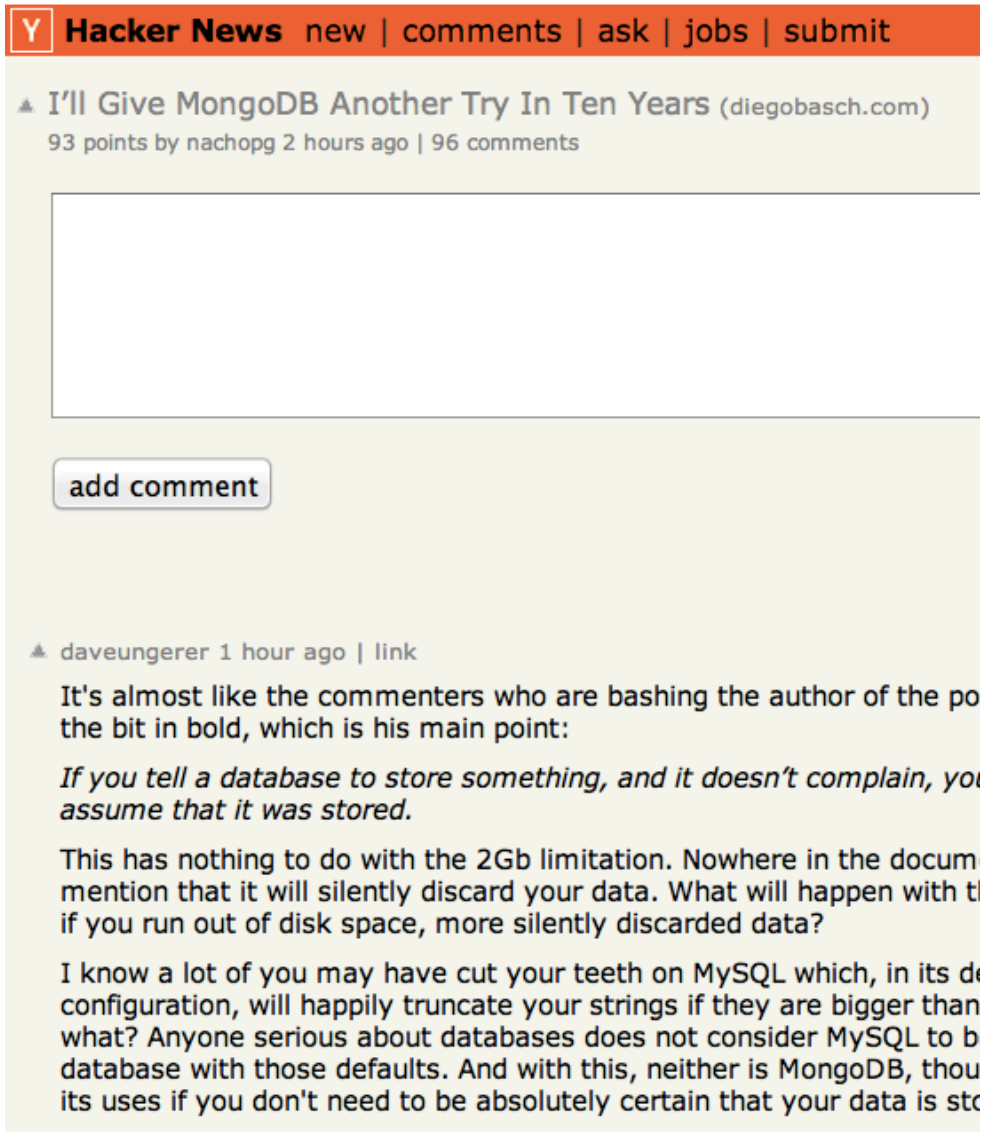
Problem: Applications Look Up Data in Different Ways

Post Page

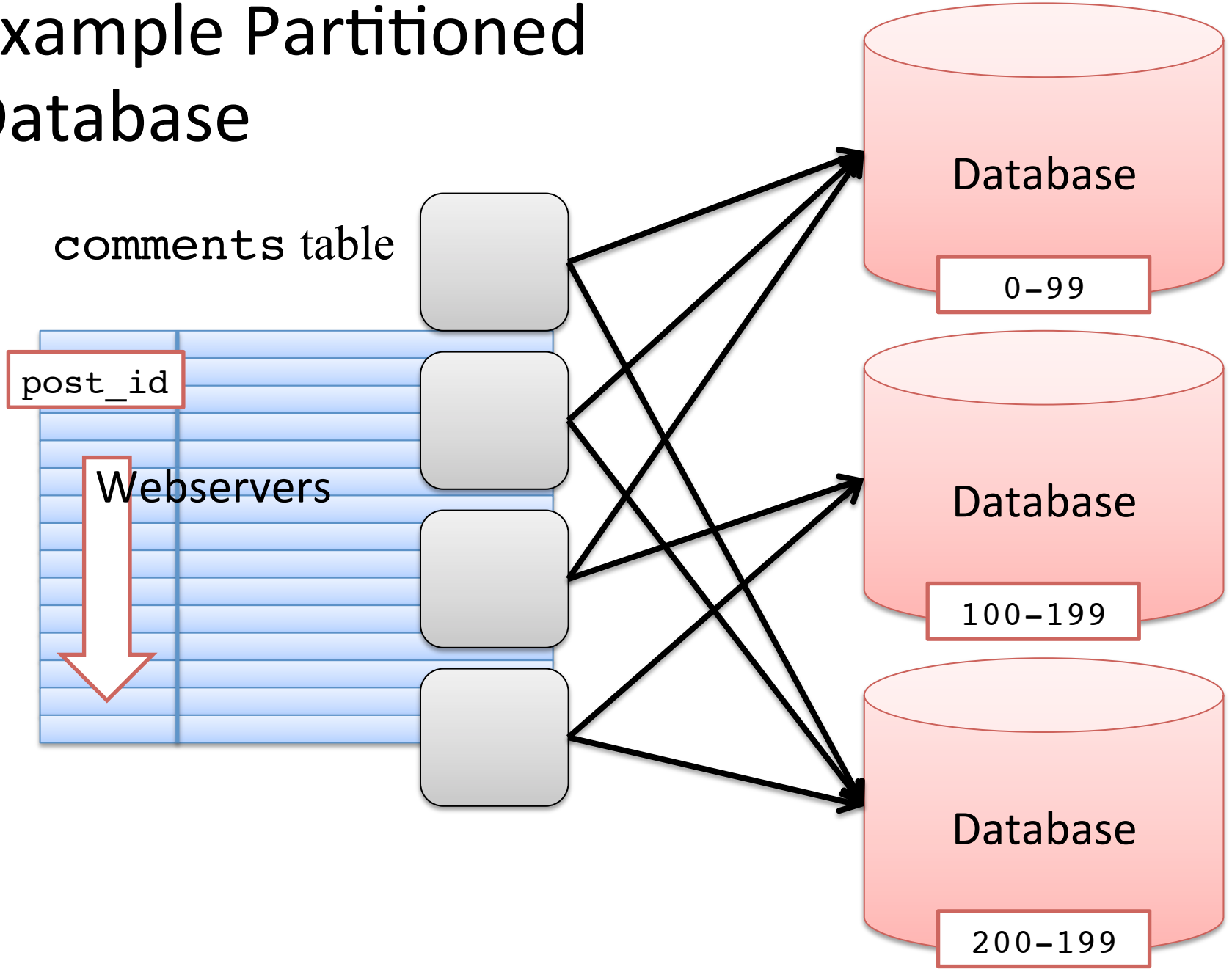
HA!

```
SELECT *
FROM comments
WHERE post_id = 100
```

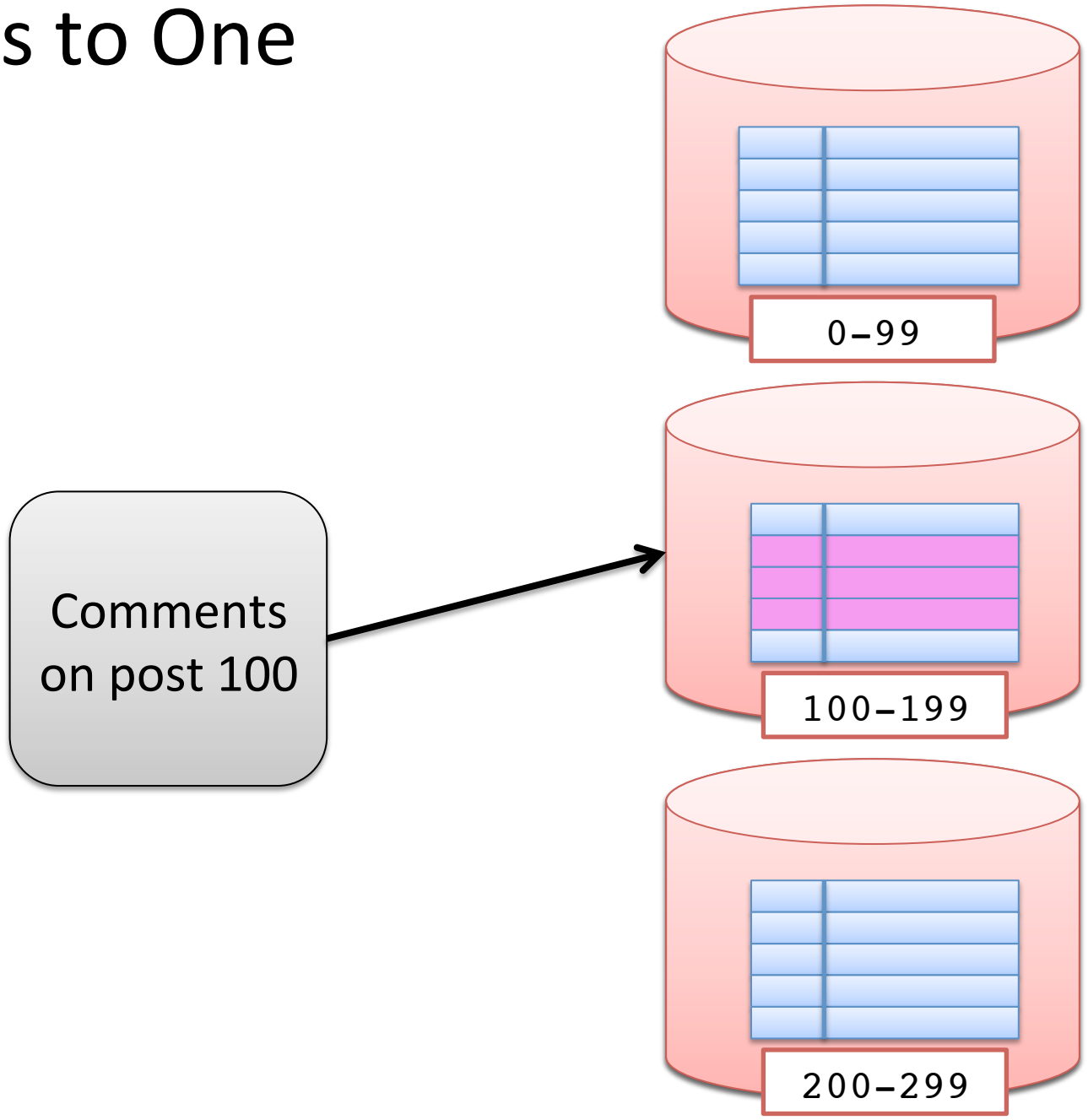
```
zrange comments:100 0 -1
```



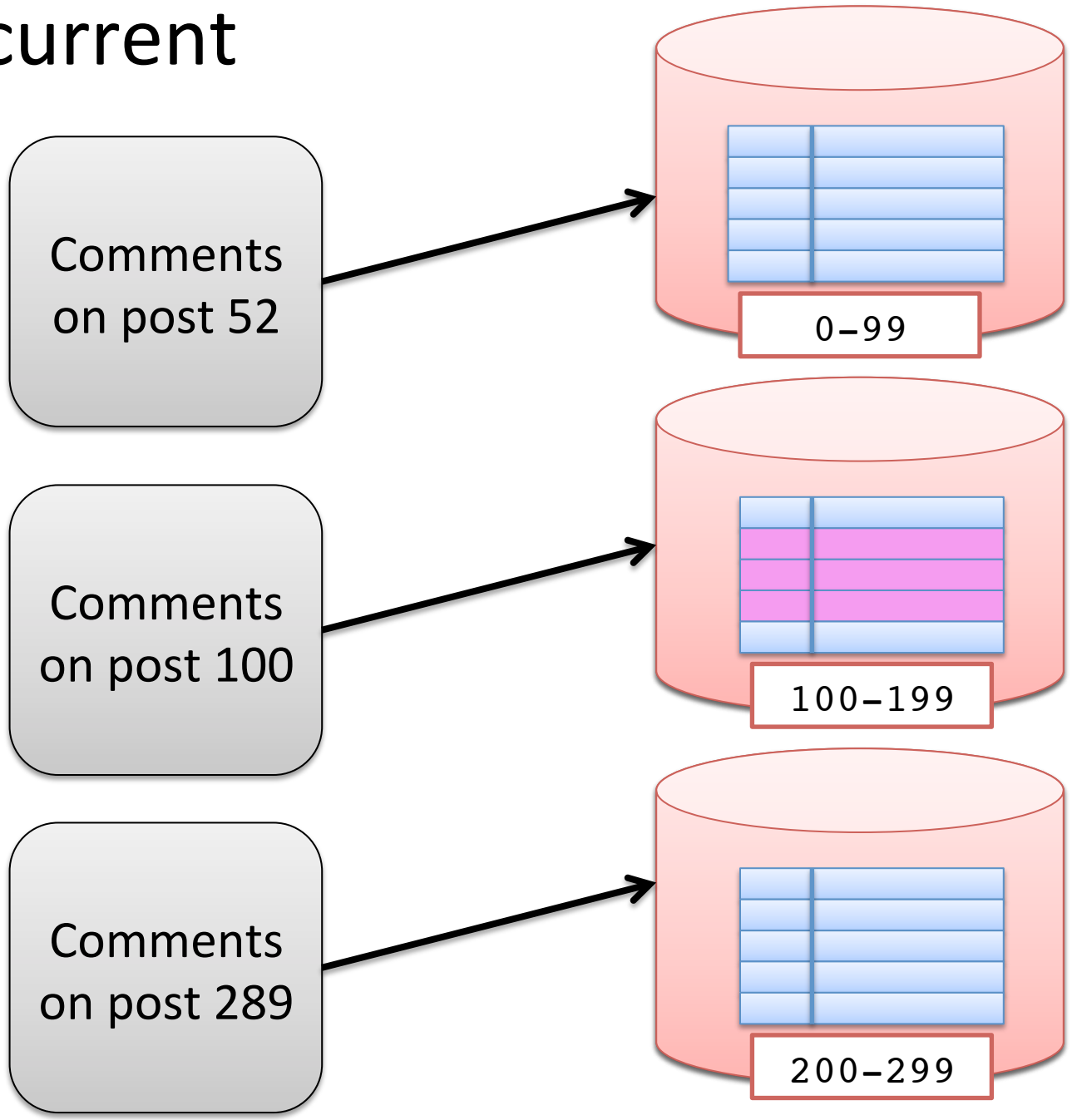
Example Partitioned Database



Query Goes to One Partition



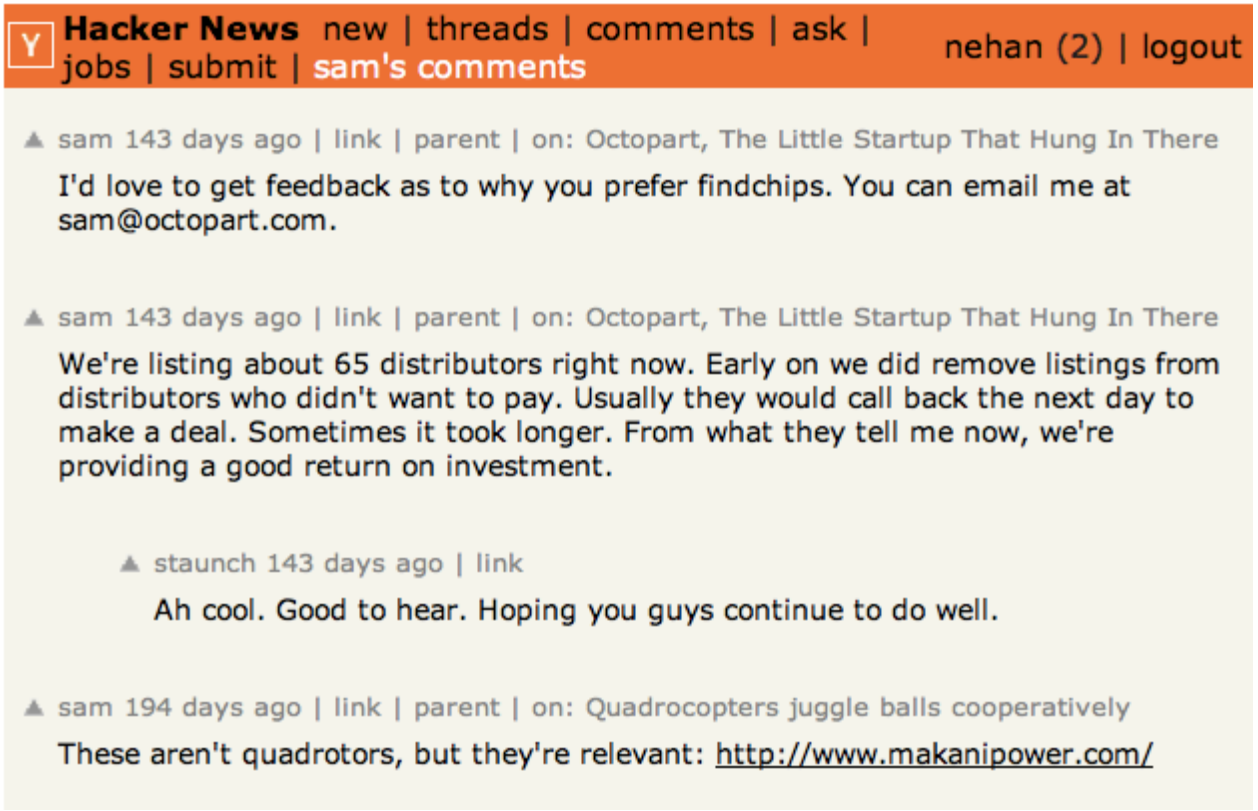
Many Concurrent Queries



User Comments Page

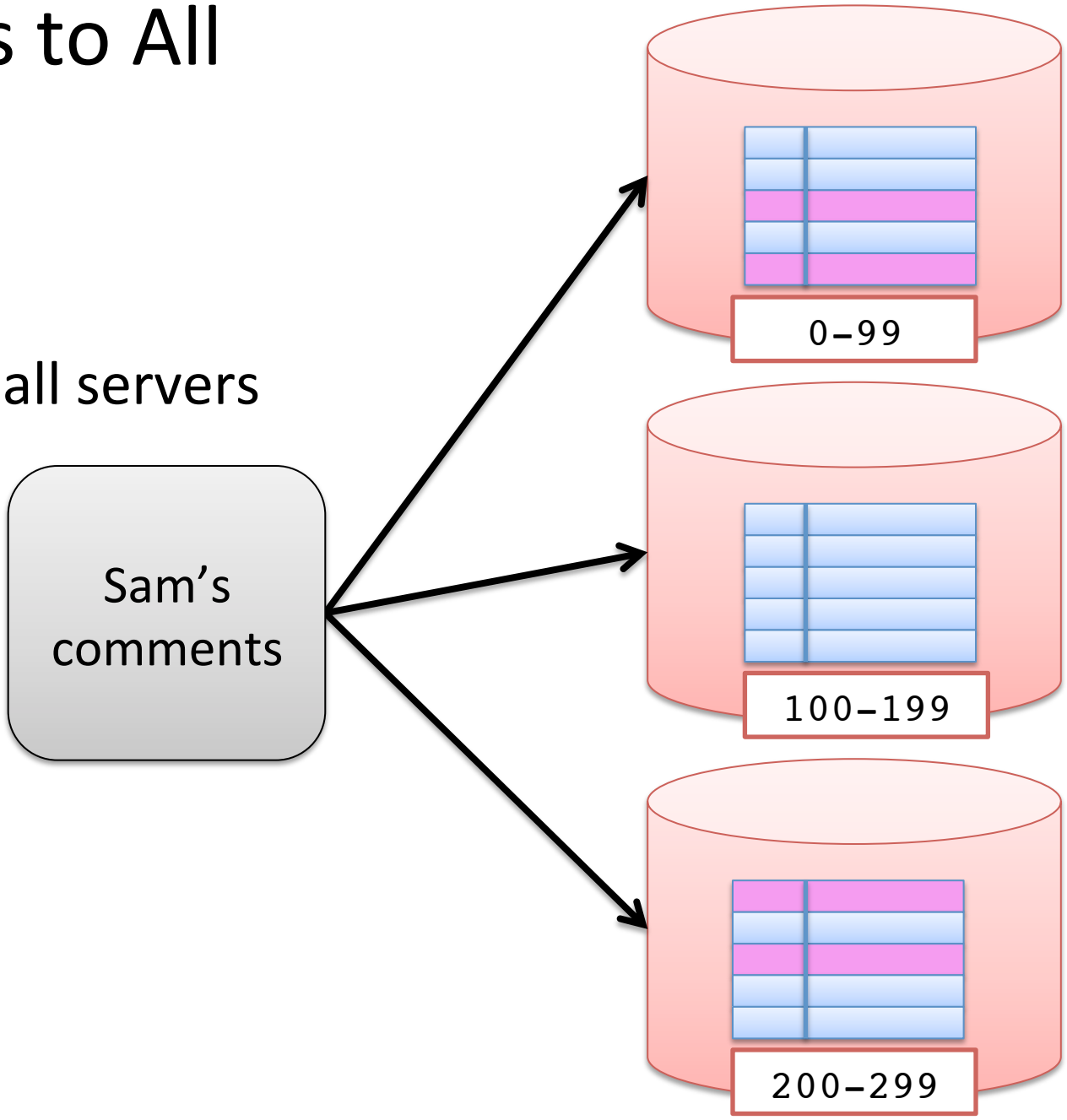
Fetch all of a user's comments:

```
SELECT * FROM comments WHERE user = 'sam'
```

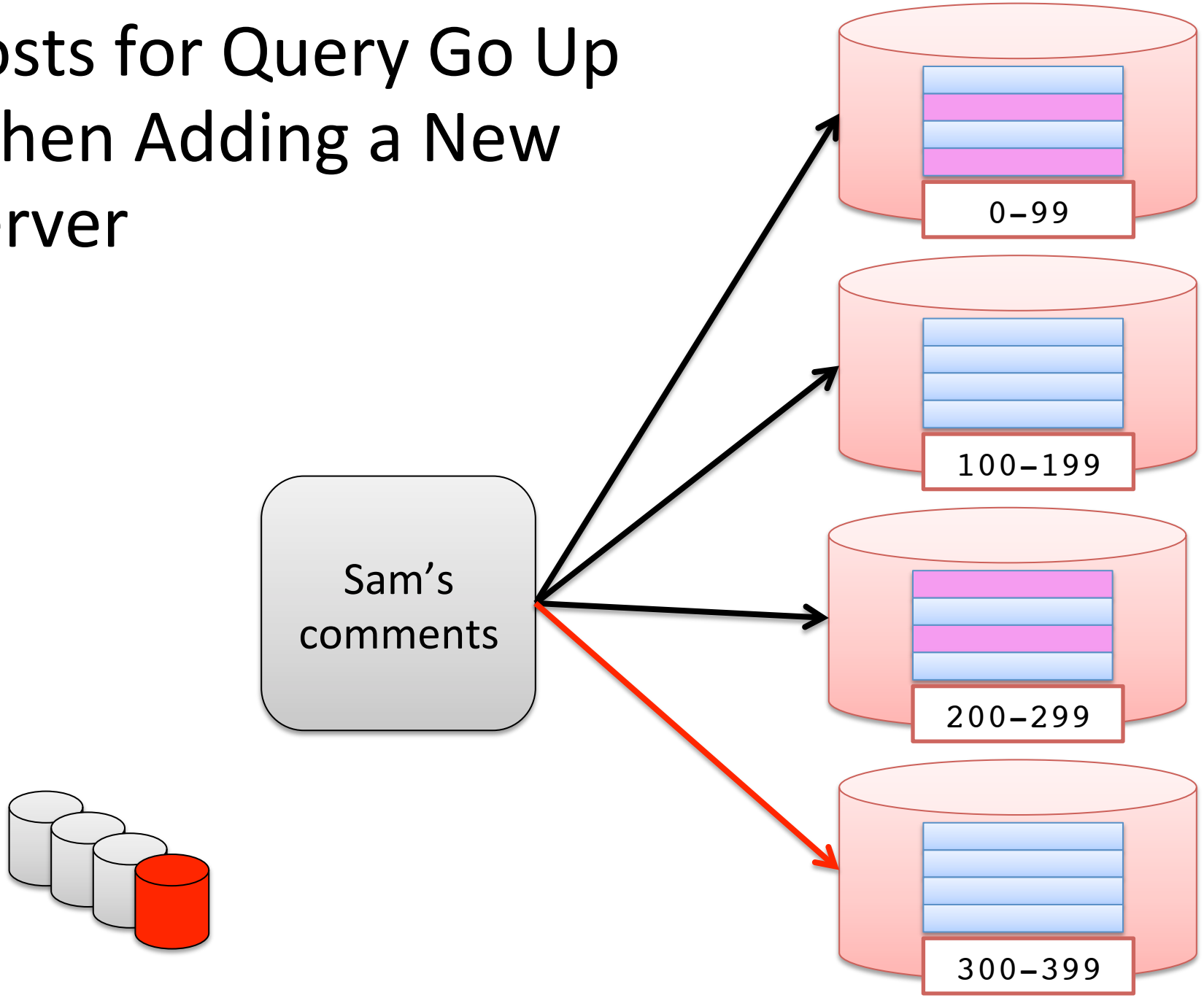


Query Goes to All Partitions

Query goes to all servers

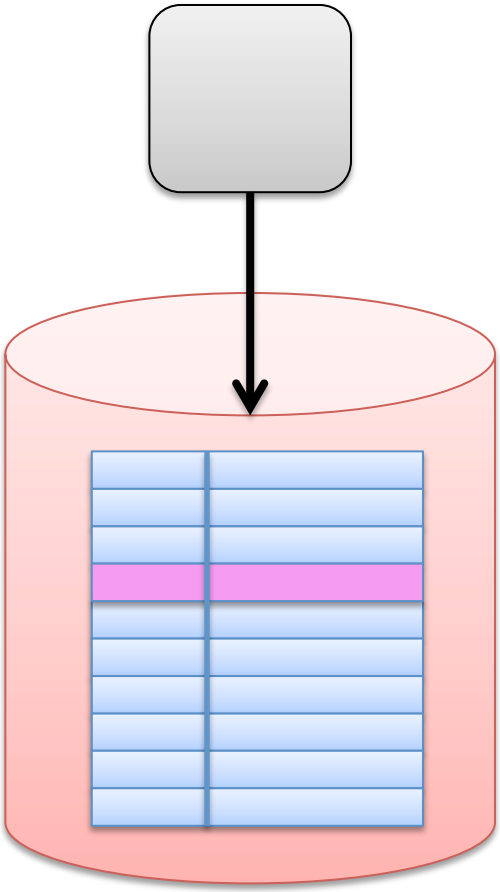


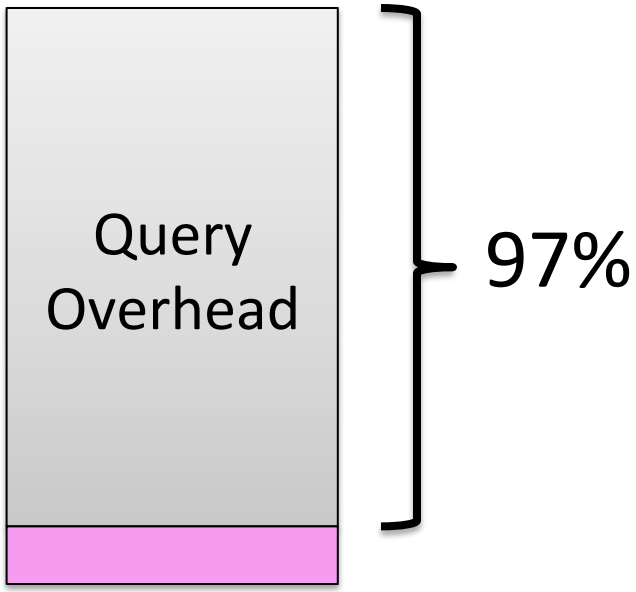
Costs for Query Go Up When Adding a New Server



CPU Cost on Server of Retrieving One Row

Two costs: Query overhead and row retrieval





Idea: Multiple Partitionings

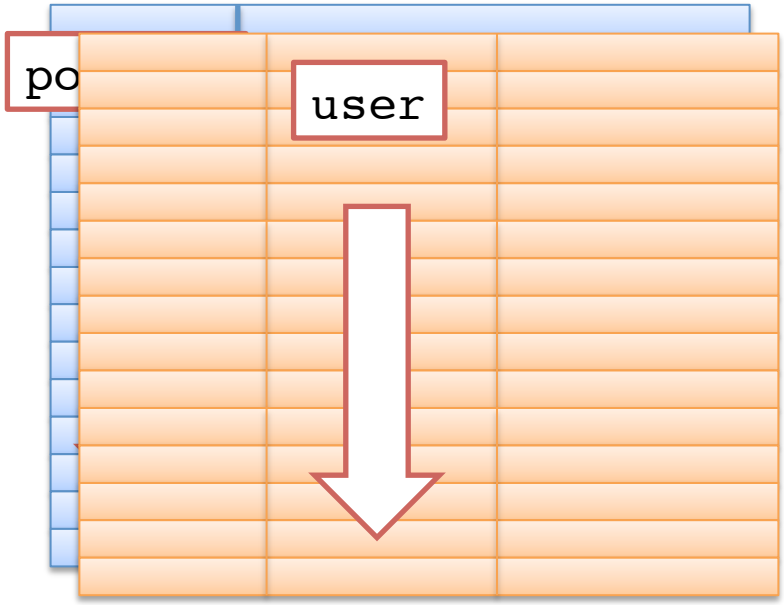
Partition comments table on `post_id` *and* `user`.

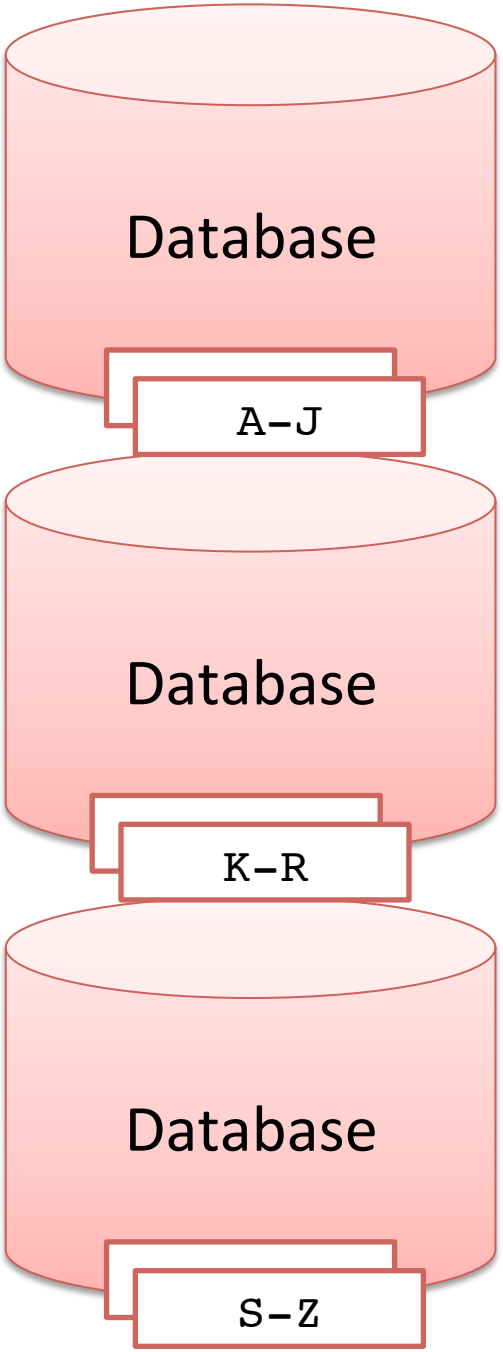
Reads can choose appropriate copy so queries go to only one server.

Writes go to all copies.

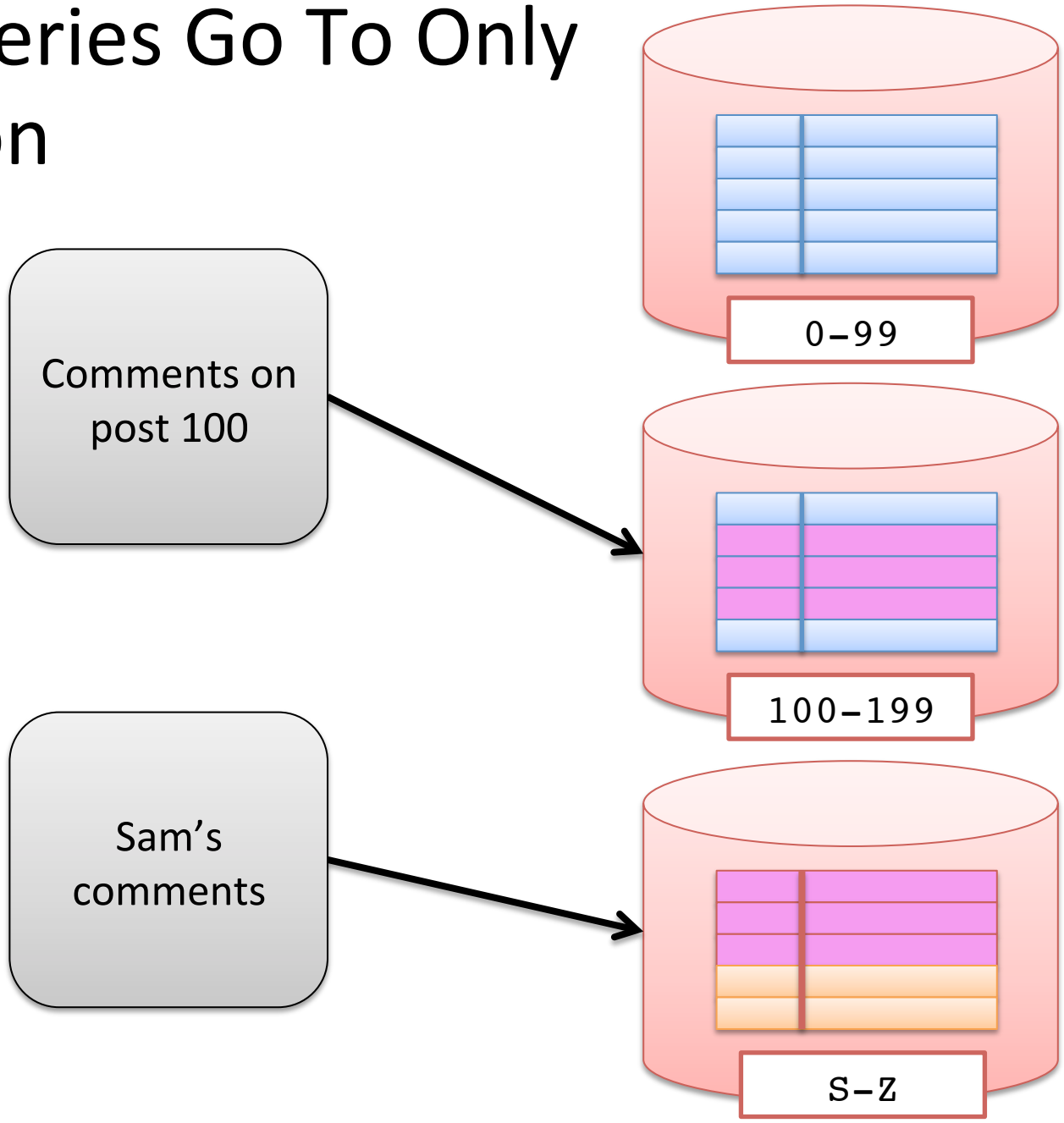
Multiple Partitionings

`comments` table

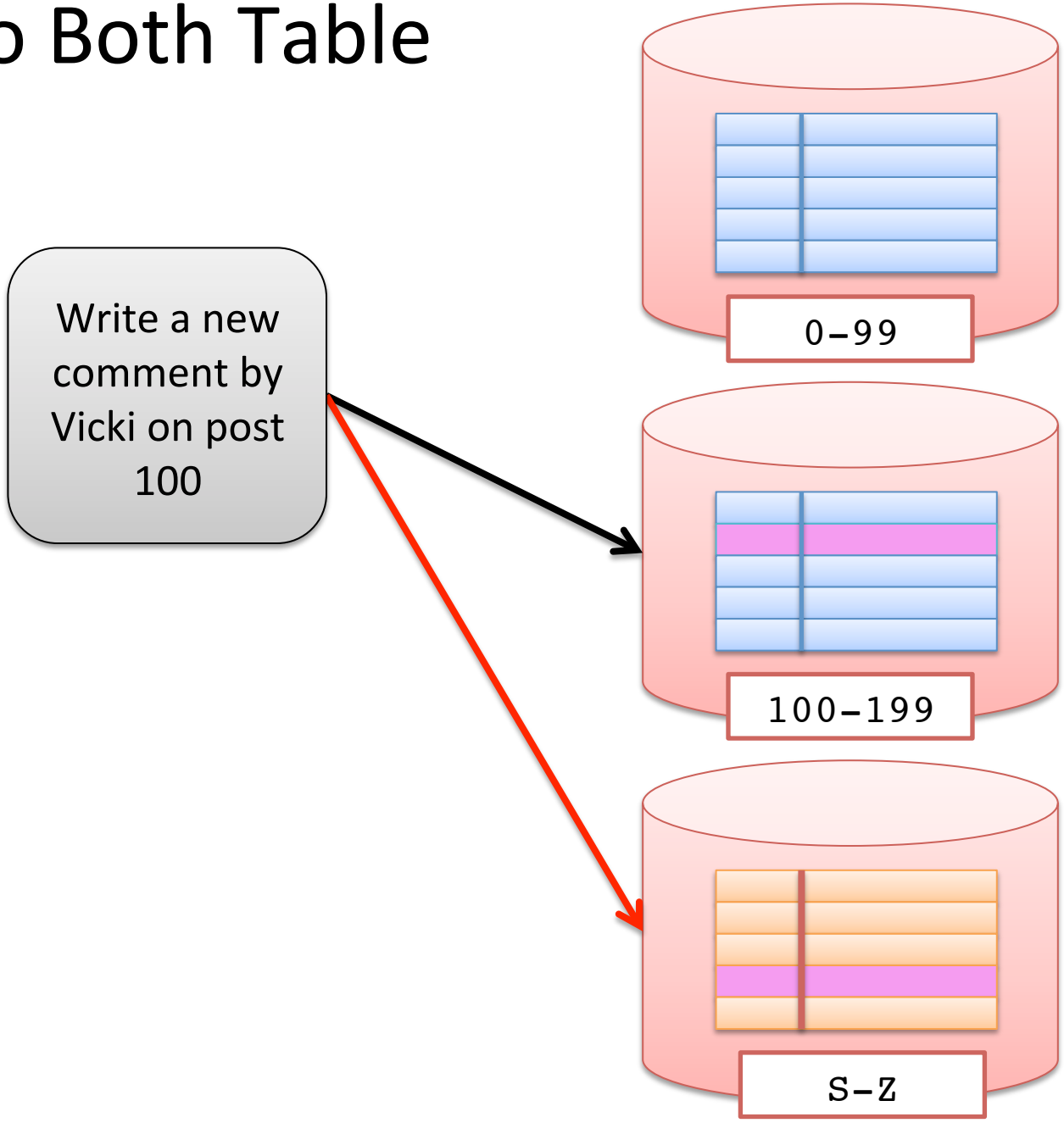




All Read Queries Go To Only One Partition



Writes Go to Both Table Copies



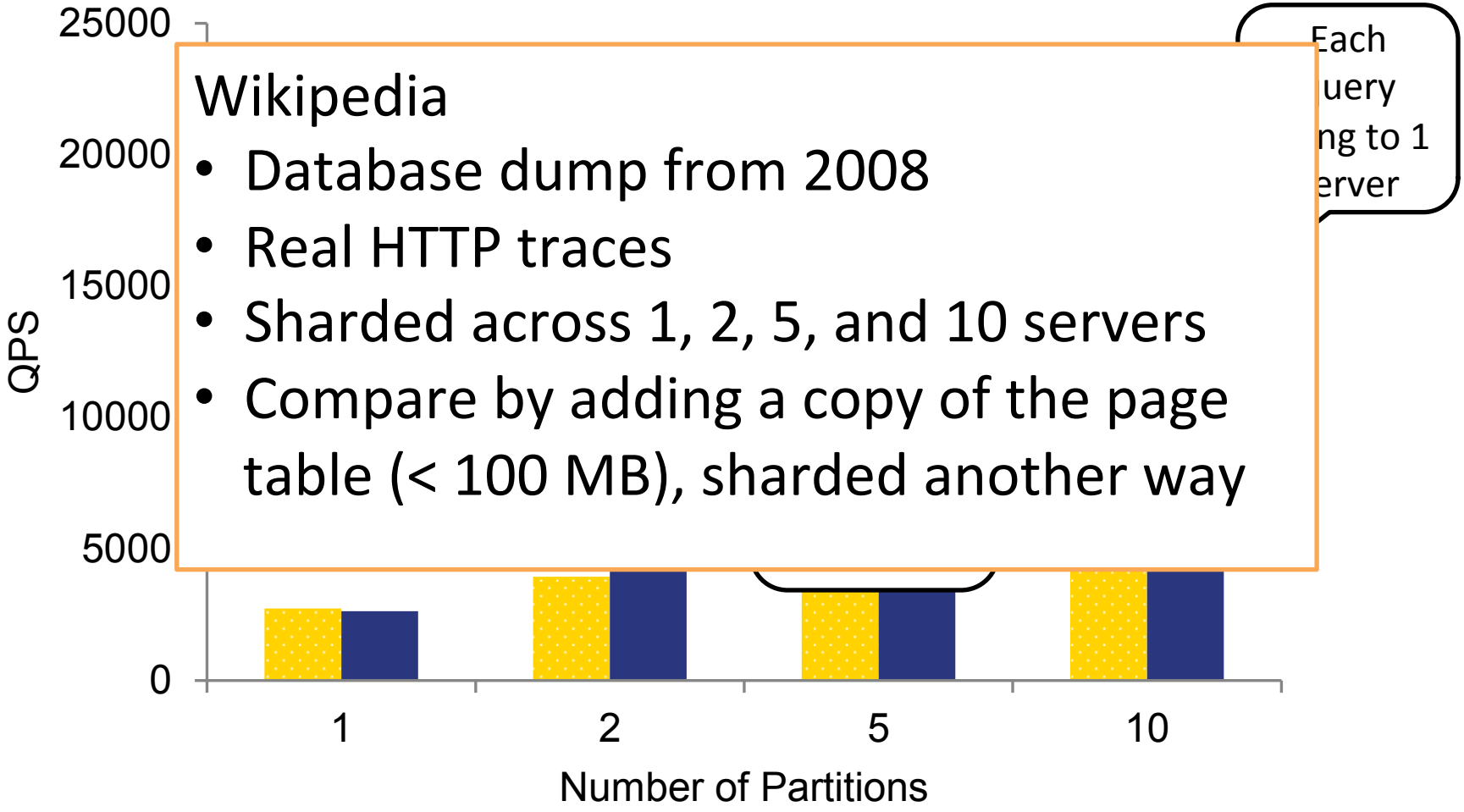
Scaling

- Reads scale by **N** – the number of servers
- Writes are slowed down by **T** – the number of table copies
- Big win if **N** >> **T**
- Table copies use more memory
  - Often don't have to copy large tables – usually metadata
- How to execute these queries?

Dixie

- Is a query planner which executes SQL queries over a partitioned database
- Optimizes read queries to minimize query overhead and data retrieval costs
- Uses a novel cost formula which takes advantage of data partitioned in different ways

Wikipedia Workload with Dixie



Wikipedia

- Database dump from 2008
- Real HTTP traces
- Sharded across 1, 2, 5, and 10 servers
- Compare by adding a copy of the page table (< 100 MB), sharded another way

Myths of Sharded Datastores

- NoSQL scales better than a relational database
- You can't do a JOIN on a sharded datastore

MYTH: You can't execute a JOIN on a sharded database

- What are the JOIN keys? What are your partition keys?
- Bad to move lots of data
- Not bad to *lookup* a few pieces of data
- Index lookup JOINs expressed as primary key lookups are fast

Join Query

Fetch all of Alice's comments on Max's posts.

`comments` table

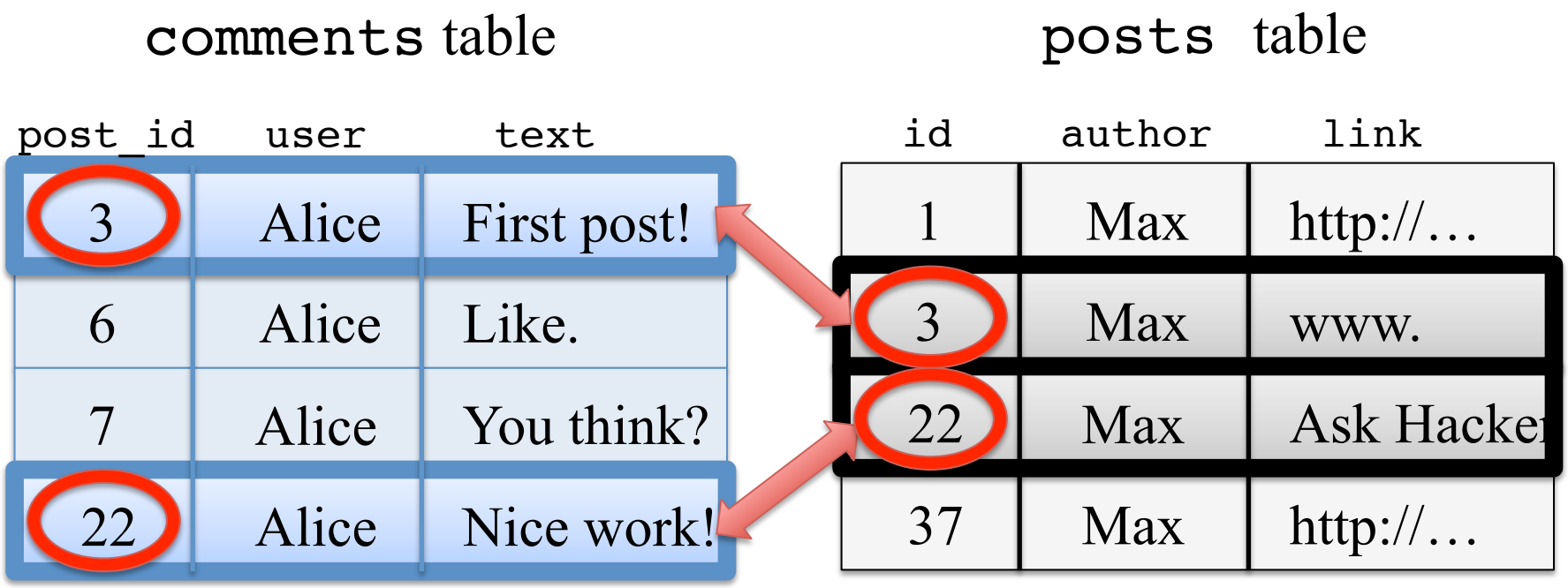
post_id	user	text
3	Alice	First post!
6	Alice	Like.
7	Alice	You think?
22	Alice	Nice work!

`posts` table

id	author	link
1	Max	http://…
3	Max	www.
22	Max	Ask Hacker
37	Max	http://…

Join Query

Fetch all of Alice's comments on Max's posts.



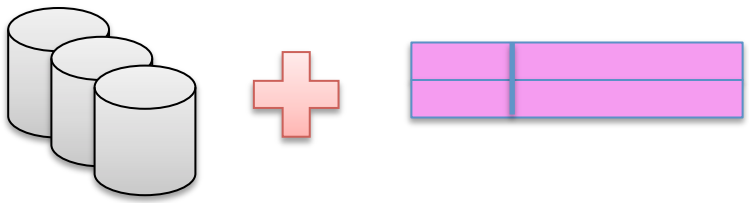
Distributed Query Planning

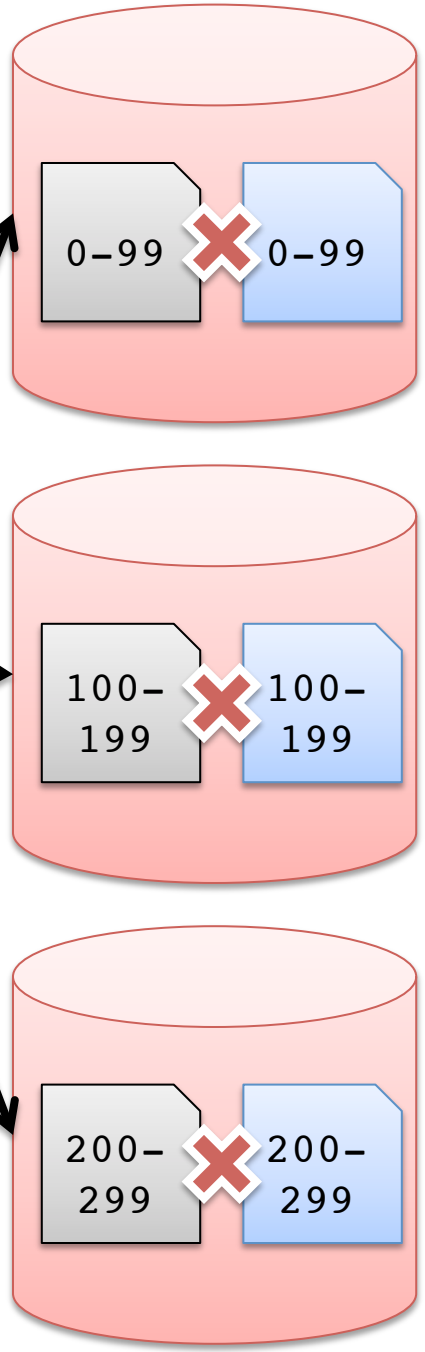
- R*
- Shore
- The State of the Art in Distributed Query Processing, by Donald Kossman
- Gizzard

Conventional Wisdom

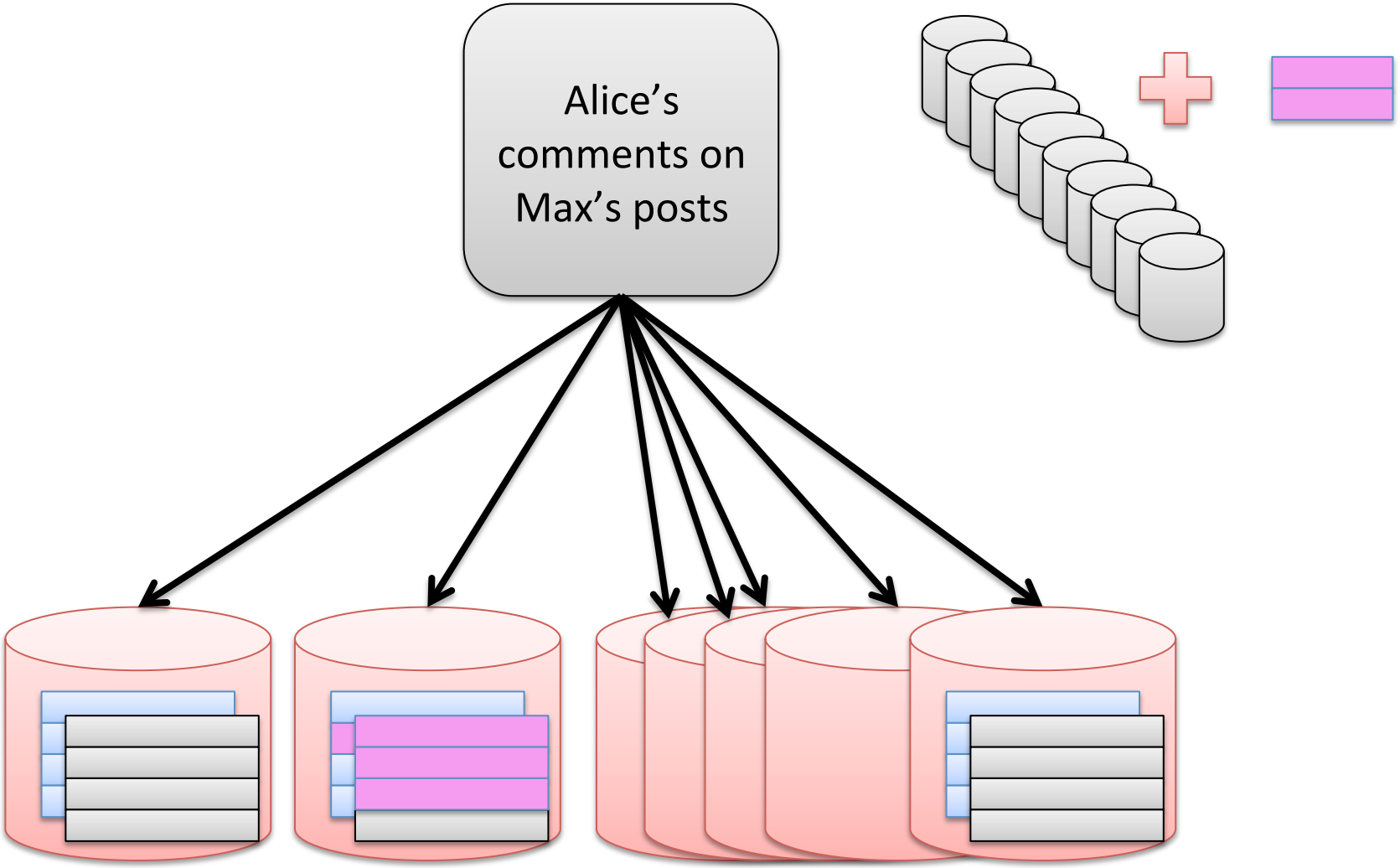
Partition on JOIN keys, Send query as is to each server.

```
SELECT   comments.text
FROM     posts, comments
WHERE    comments.post_id = posts.id
AND      posts.author = 'Max'
AND      comments.user = 'Alice'
```



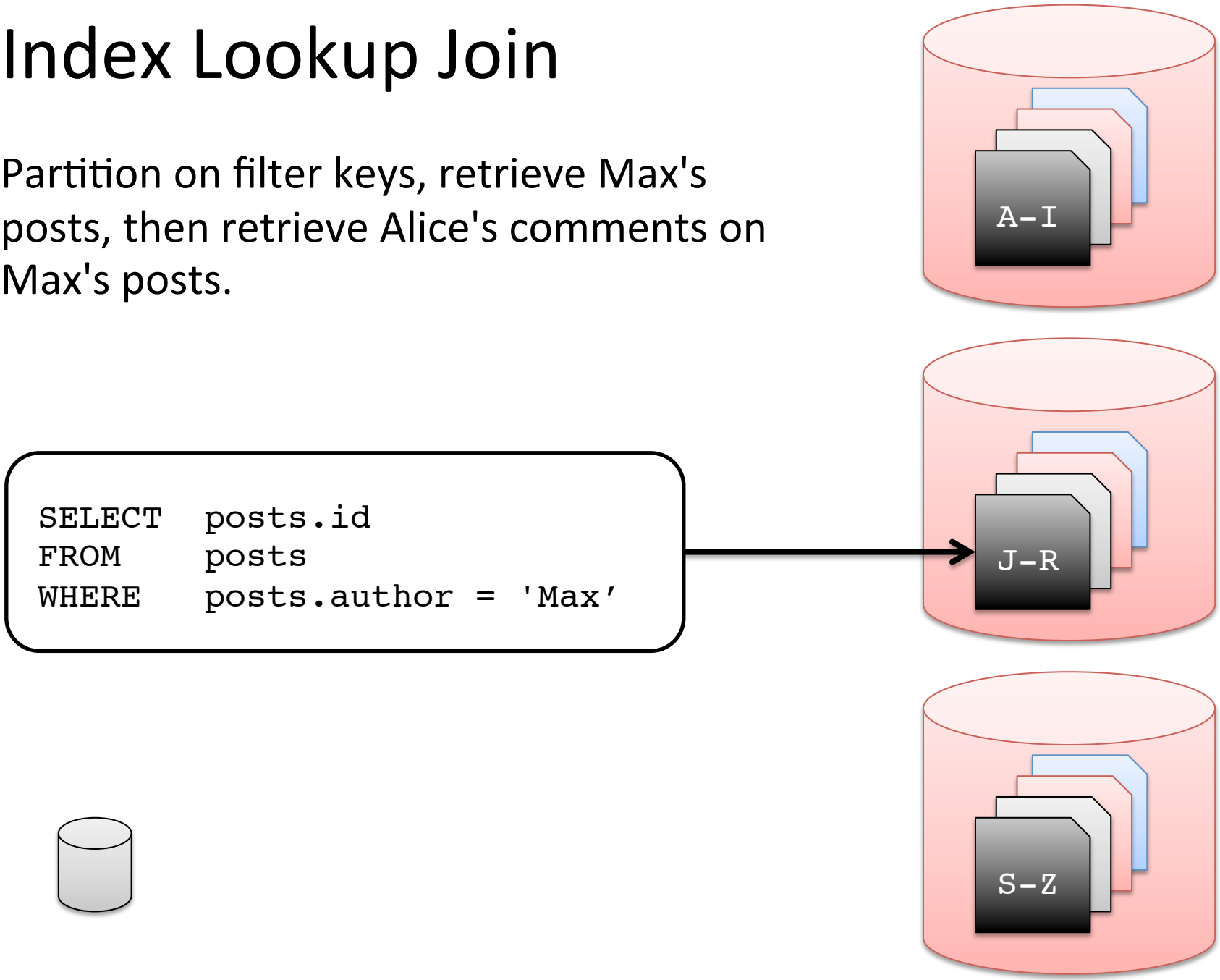


Conventional Plan Scaling



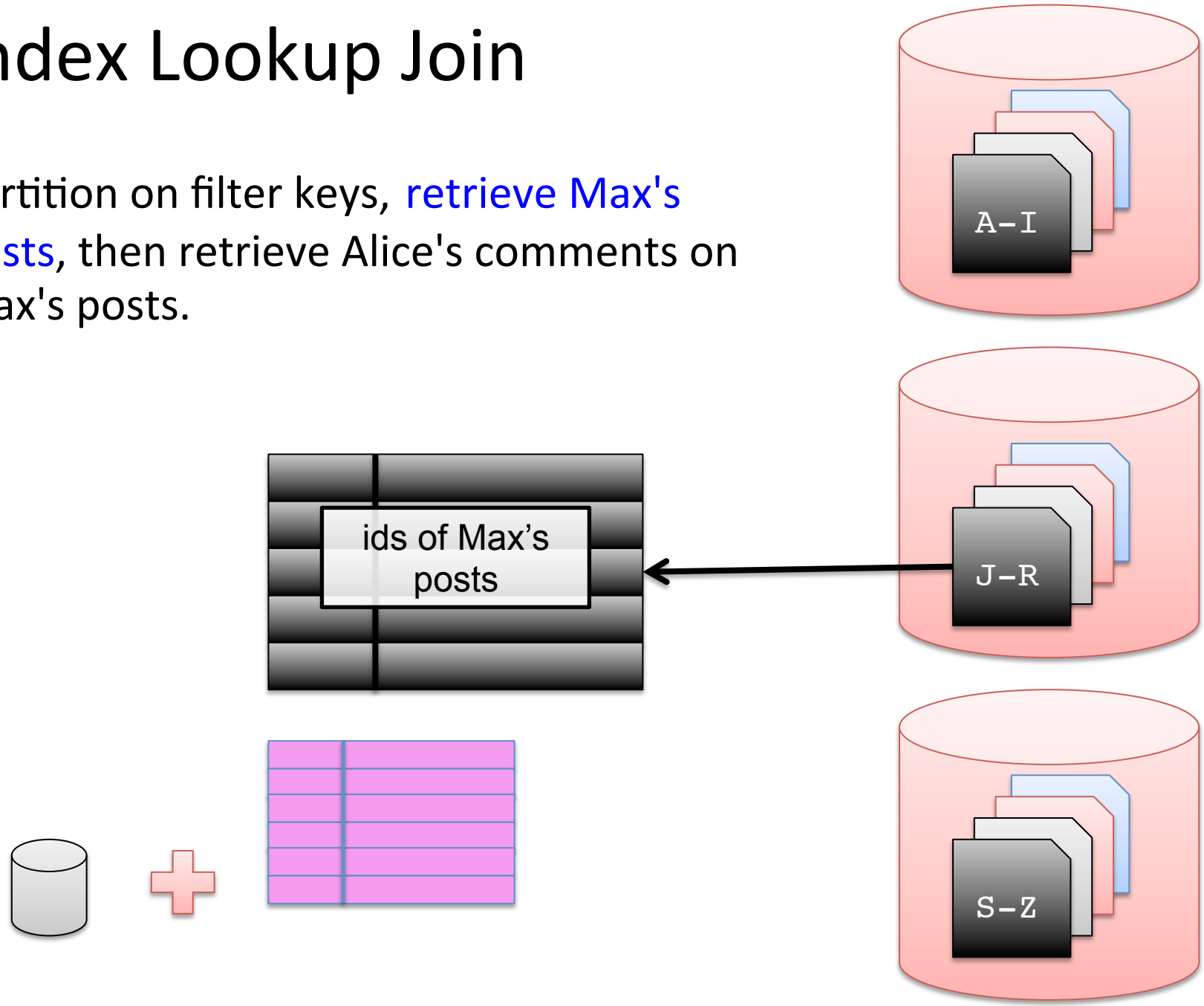
Index Lookup Join

Partition on filter keys, retrieve Max's posts, then retrieve Alice's comments on Max's posts.



Index Lookup Join

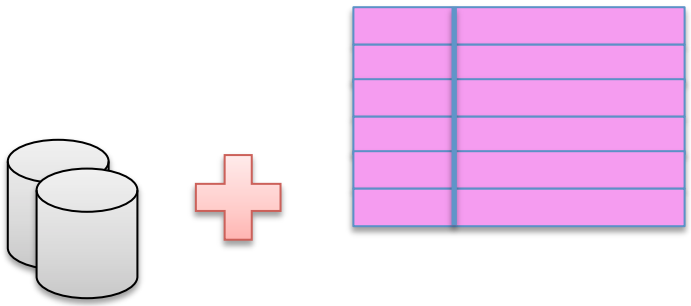
Partition on filter keys, retrieve Max's posts, then retrieve Alice's comments on Max's posts.

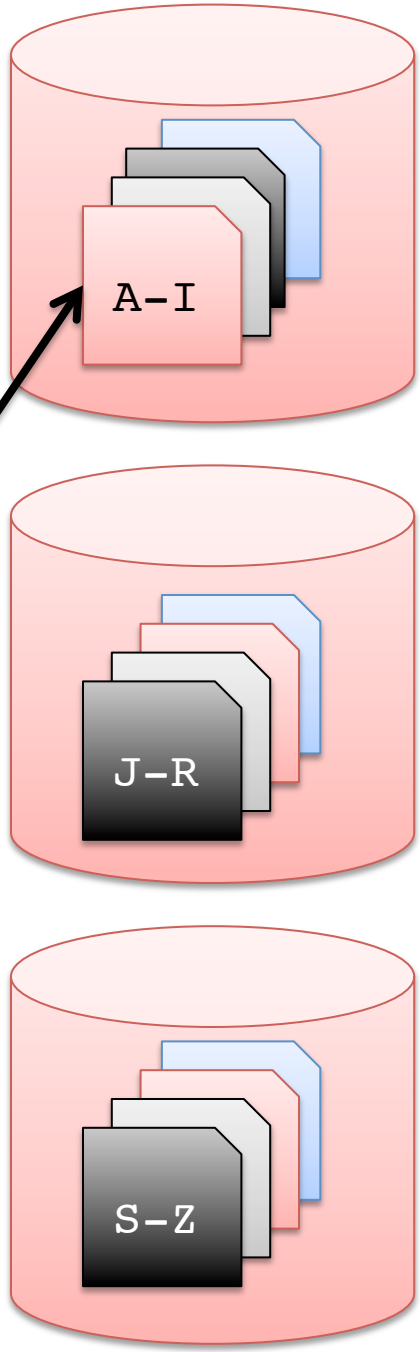


Index Lookup Join

Partition on filter keys, retrieve Max's posts, then retrieve Alice's comments on Max's posts.

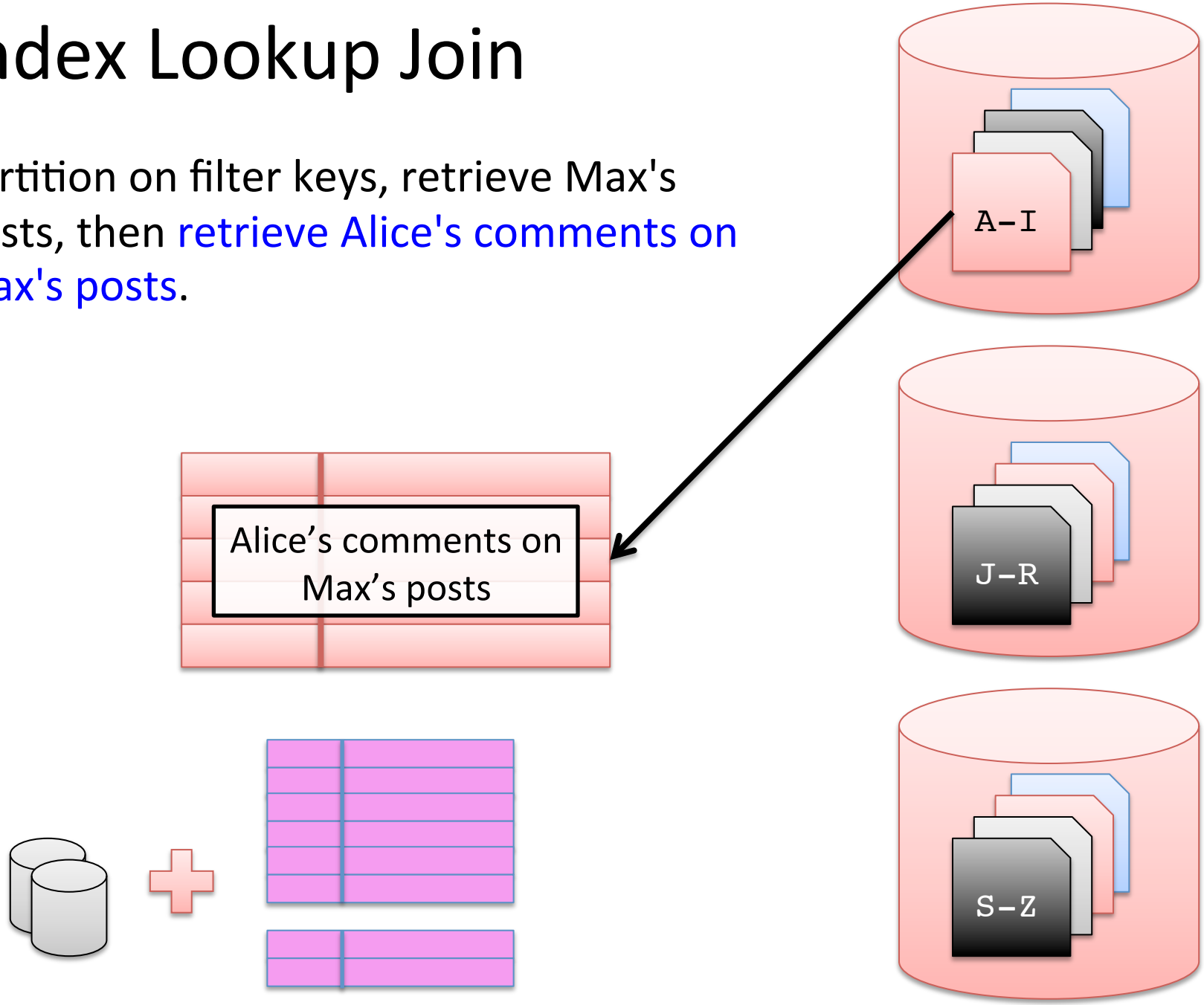
```
SELECT   comments.text
FROM     comments
WHERE    comments.post_id IN […]
AND      comments.user = 'Alice'
```



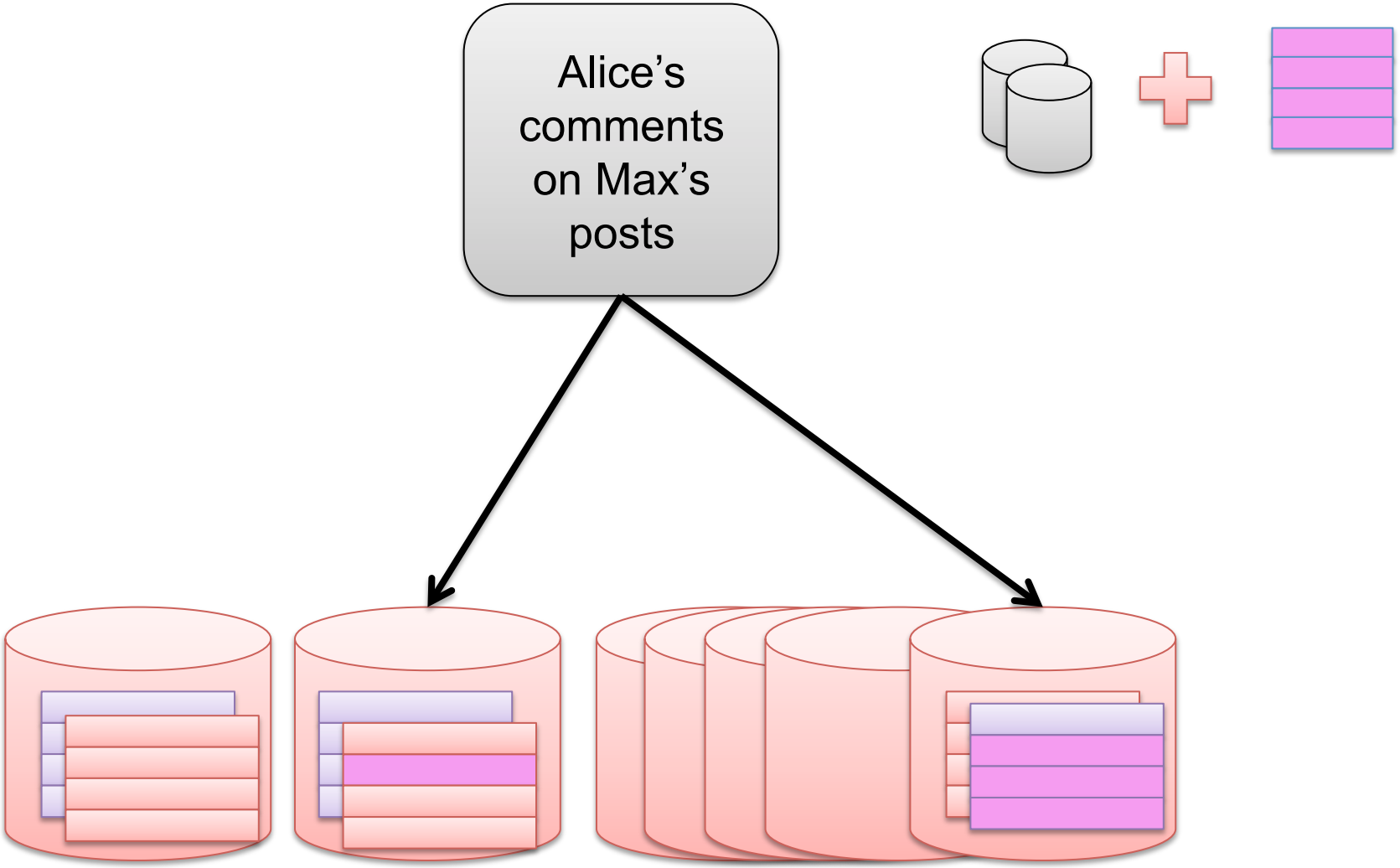


Index Lookup Join

Partition on filter keys, retrieve Max's posts, then retrieve Alice's comments on Max's posts.

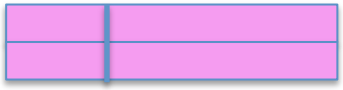


Index Lookup Join Plan Scaling

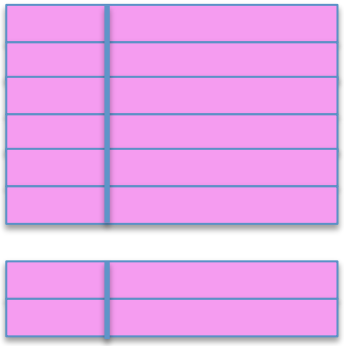


Comparison

Conventional Plan



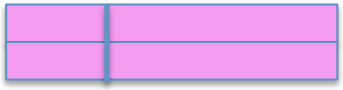
Index Lookup Plan



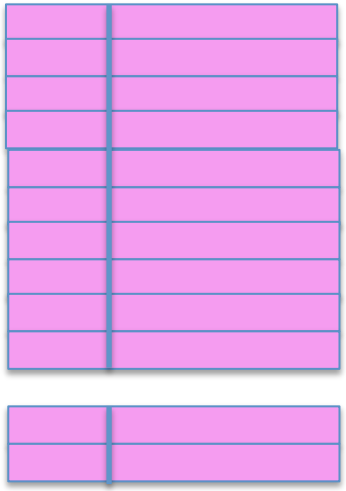
Intermediate data, Max's posts Alice DIDN'T comment on

Comparison

Conventional Plan



Index Lookup Plan

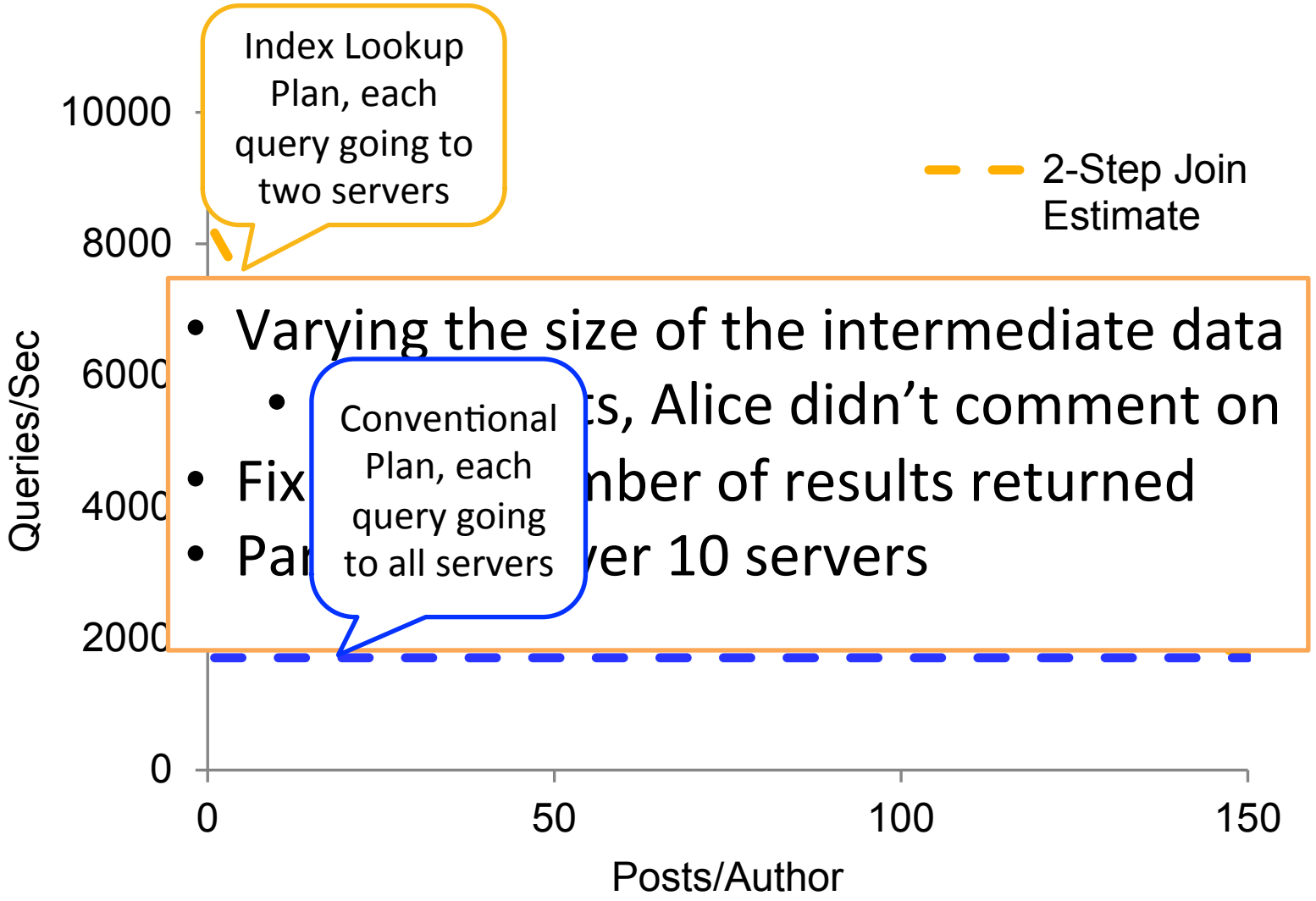


Intermediate data, Max's posts Alice DIDN'T comment on

Dixie

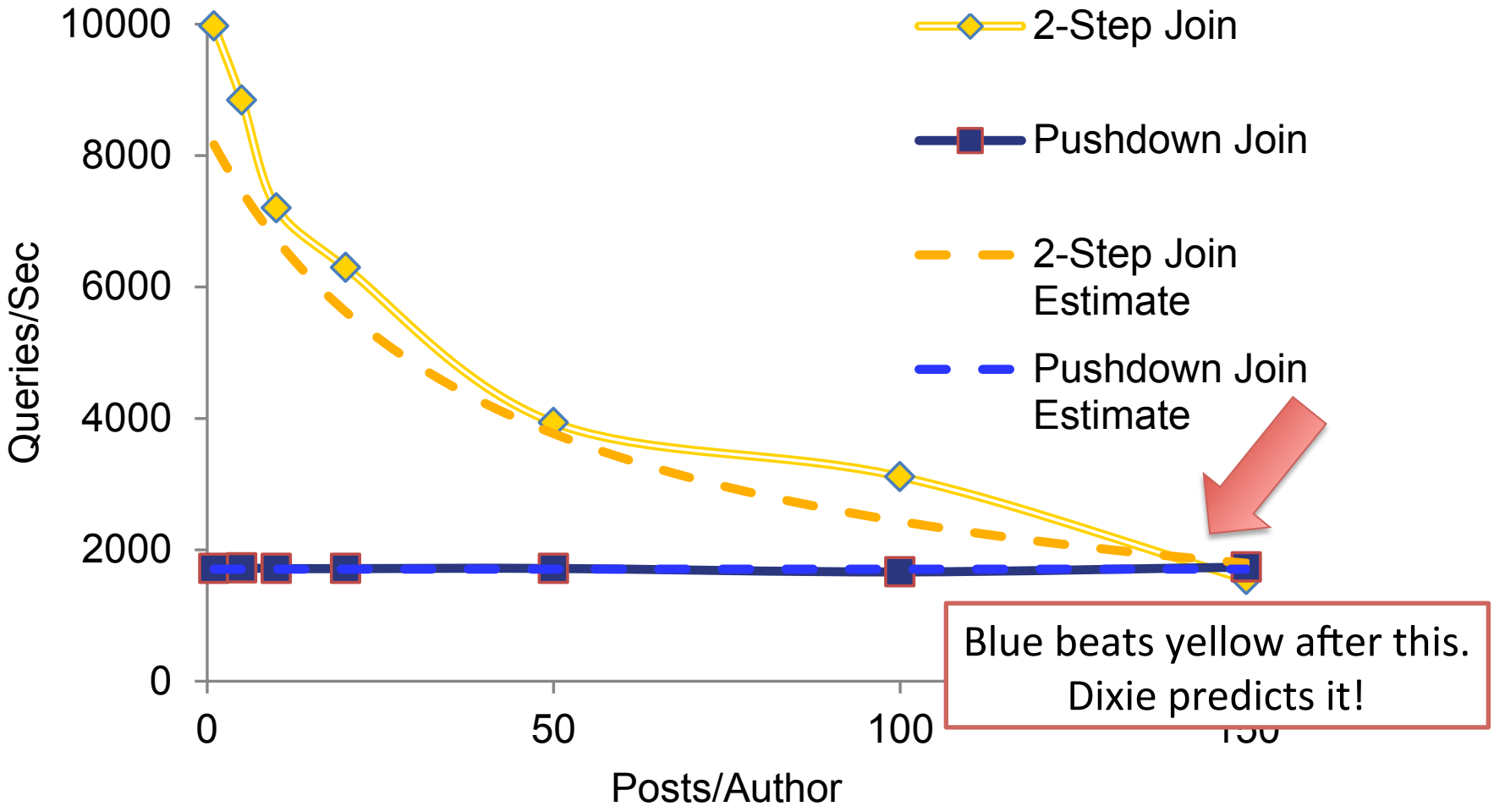
- Cost model and query optimizer which predicts the costs of the two plans
- Query executor which executes the original query, designed for a single database, on a partitioned database with table copies

Dixie's Predictions for the Two Plans



- Varying the size of the intermediate data
  - ts, Alice didn't comment on
- Fix nber of results returned
- Par ver 10 servers

Performance of the Two Plans



JOINs Are Not Evil

- When only transferring small amounts data
- And with carefully partitioned data

Lessons Learned

- Don’t worry at the beginning: Use what you know
- Not about SQL vs. NoSQL systems – you can scale any system
- More about complex vs. simple queries
- When you do scale, try to make every query use one (or a few) shards

Research: Caching

- Applications use a cache to store results computed from a webserver or database rows
- Expiration or invalidation?
- Annoying and difficult for app developers to invalidate cache items correctly

Work in progress with Bryan Kate, Eddie Kohler, Yandong Mao, and Robert Morris
Harvard and MIT

Solution: Dependency Tracking Cache

- Application indicates what data went into a cached result
- The system will take care of invalidations
- Don’t need to recalculate expired data if it has not changed
- Always get the freshest data when data has changed
- Track ranges, even if empty!

Example Application: Twitter

- Cache a user's view of their twitter homepage (timeline)
- Only invalidate when a followed user tweets
- Even if many followed users tweet, only recalculate the timeline once when read
- No need to recalculate on read if no one has tweeted

Challenges

- Invalidation + expiration times
  - Reading stale data
- Distributed caching
- Cache coherence on the original database rows

Benefits

- Application gets all the benefits of caching
  - Saves on computation
  - Less traffic to the backend database
- Doesn't have to worry about freshness

Summary

- Choosing a datastore
- Dixie
- Dependency tracking cache

Thanks!

narula@mit.edu

http://nehanaru.la

@neha