EPIC
Institute of Technology
Powered by <epam>

Lecture 3. Storages.

Course: Real-Time Backend

Lecturer: Gleb Lobanov

May, 2024

EPIC
Institute of Technology
Powered by <epam>

Contents

EPIC Institute of Technology
Powered by <epam>

01

Distributed storage. Recap

A **distributed system** is a system whose components are located on **different networked** computers, which **communicate** and **coordinate** their actions by passing messages to one another

M. van Steen and A.S. Tanenbaum, Distributed Systems, 3rd ed.

EPIC Institute of Technology
Powered by <epam>

A distributed data store is a computer network where information is stored on **more than one node**, often in a **replicated** fashion

Yaniv Pessach, Distributed Storage: Concepts, Algorithms, and Implementations

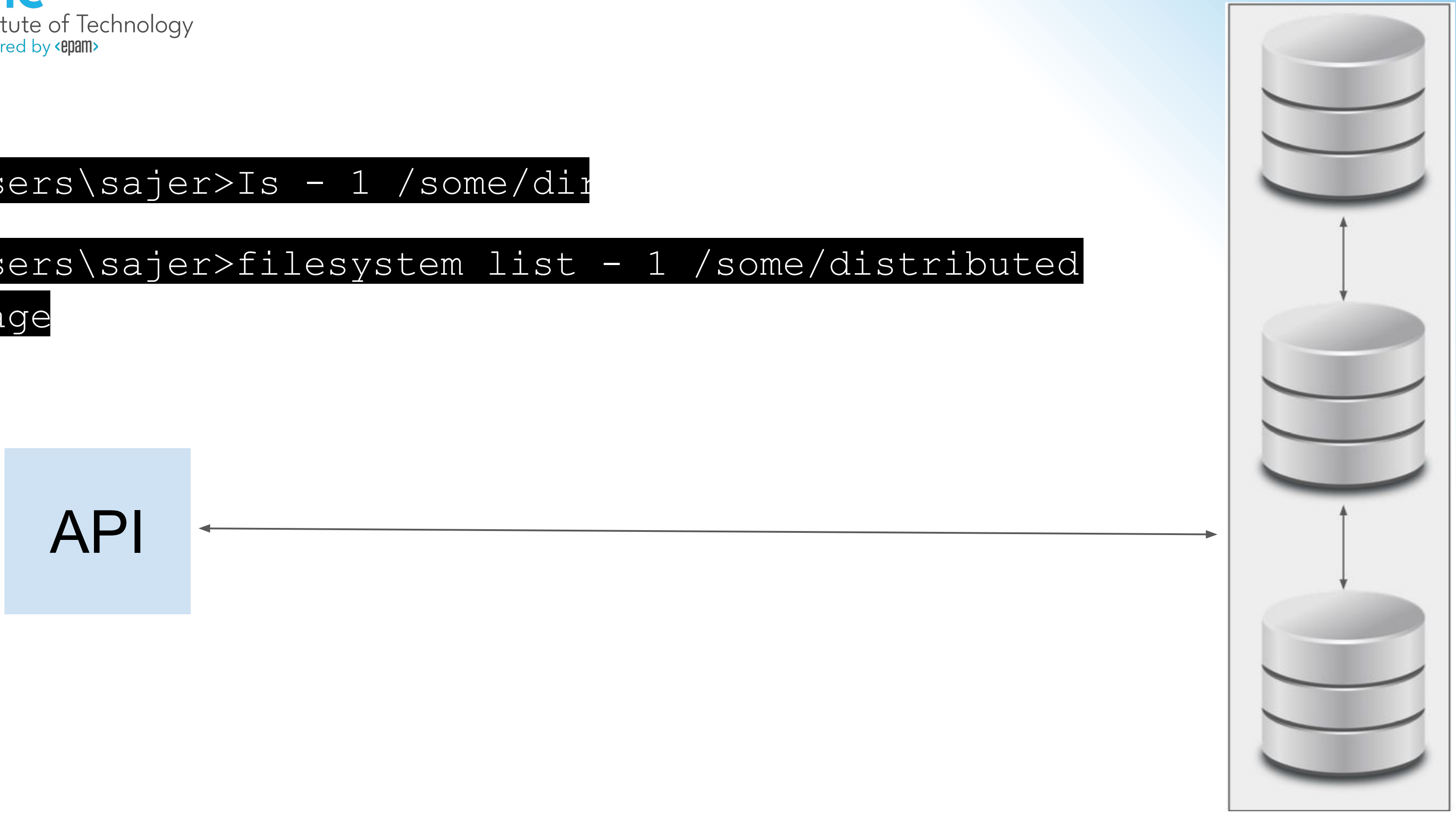
Examples

- **Storage for big websites**
- **Big data computation**
- Bank software
- …
- Infrastructure where properties of the distributed systems are desired

We also want to provide abstractions so client can work with distributed system conveniently

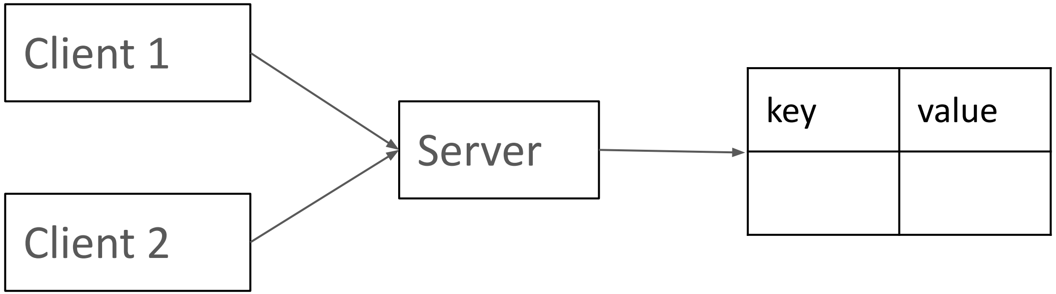
```
C:\Users\sajer>Is - 1 /some/dir
```

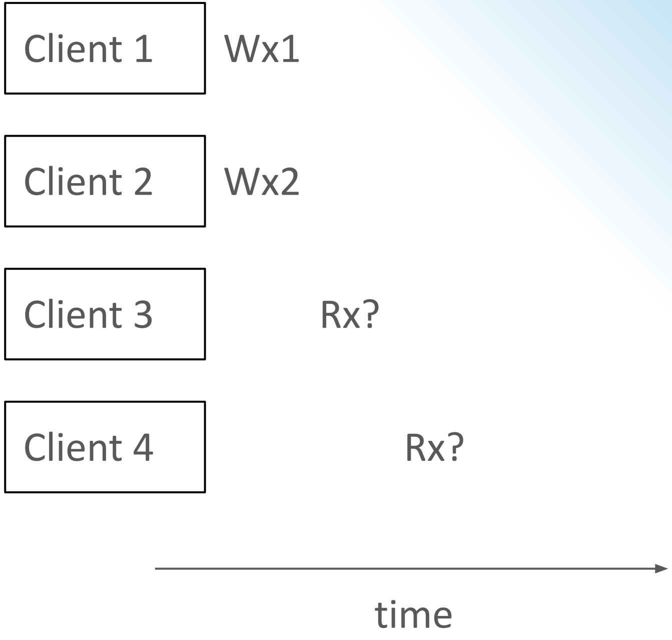
```
C:\Users\sajer>filesystem list - 1 /some/distributed storage
```



Strong consistency

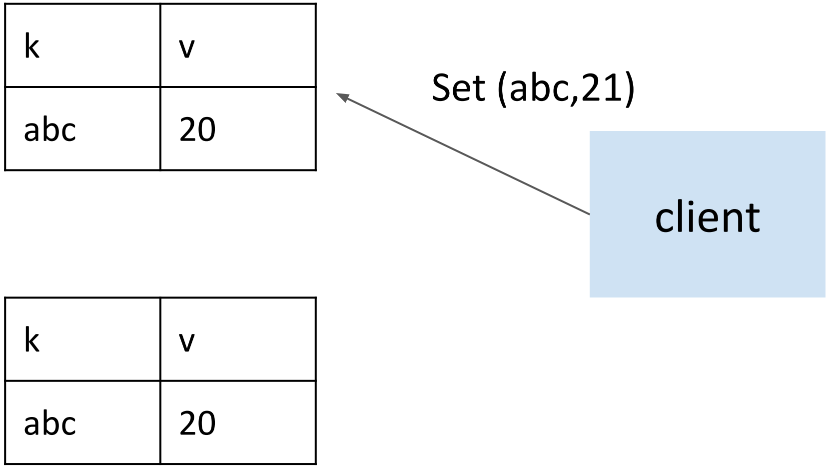
- For any interaction the system looks like all the operations are performed sequentially on a single machine
- Do we actually need such strong guarantees?





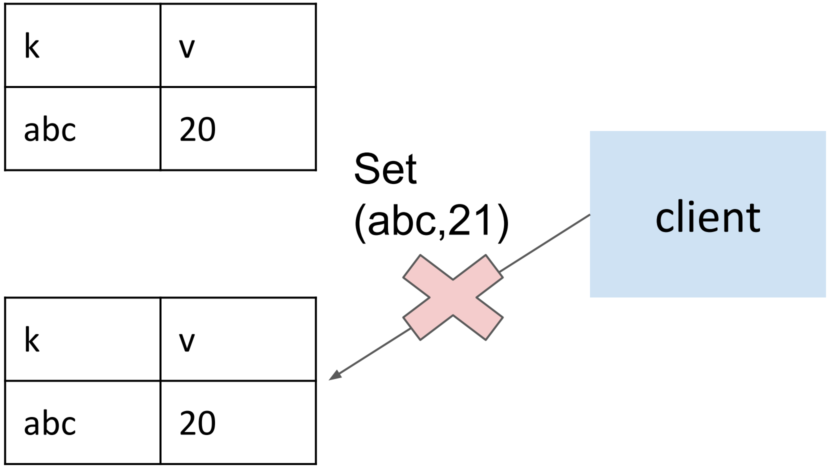
Example: KV service

- Set (k,v)
- Get (k) -> v



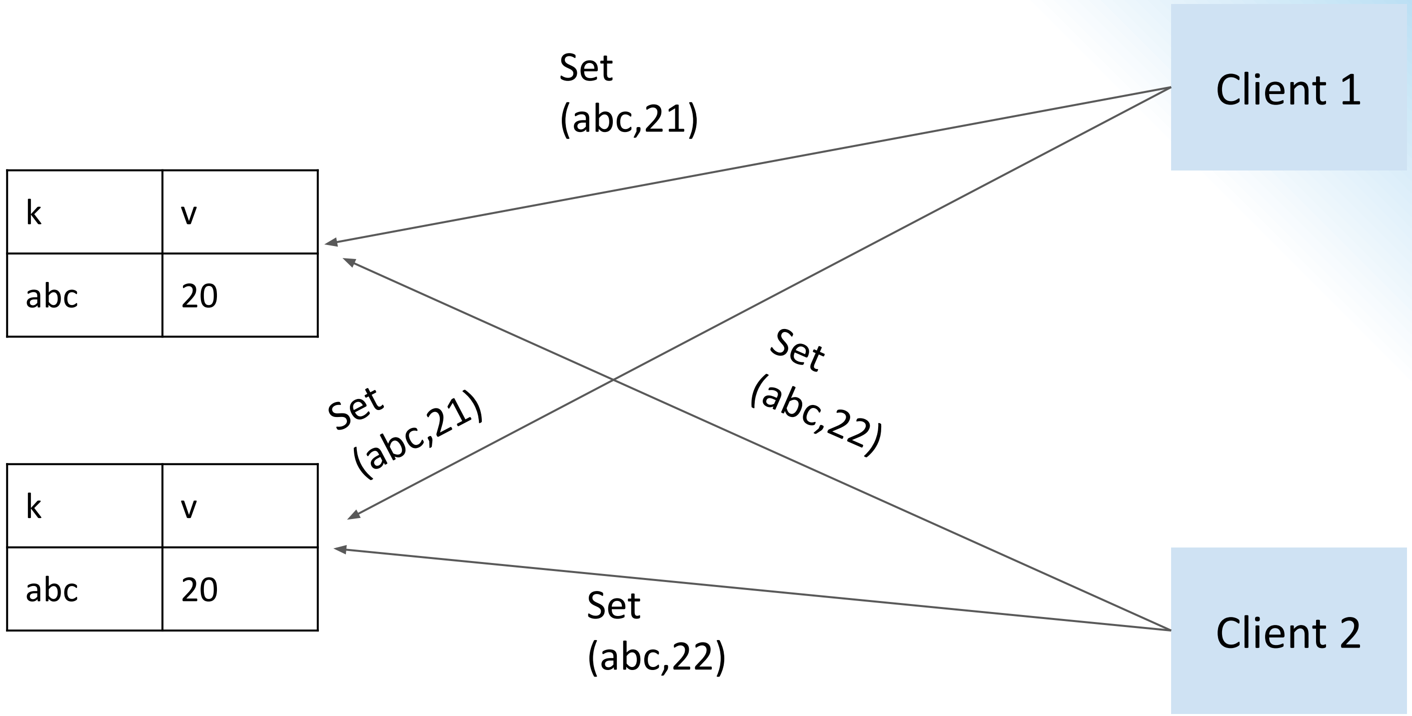
Example: KV service

- Set (k,v)
- Get (k) -> v



Example: KV service

- Set (k,v)
- Get (k) -> v



Example: KV service

- Set (k,v)
- Get (k) -> v

k	v
abc	21

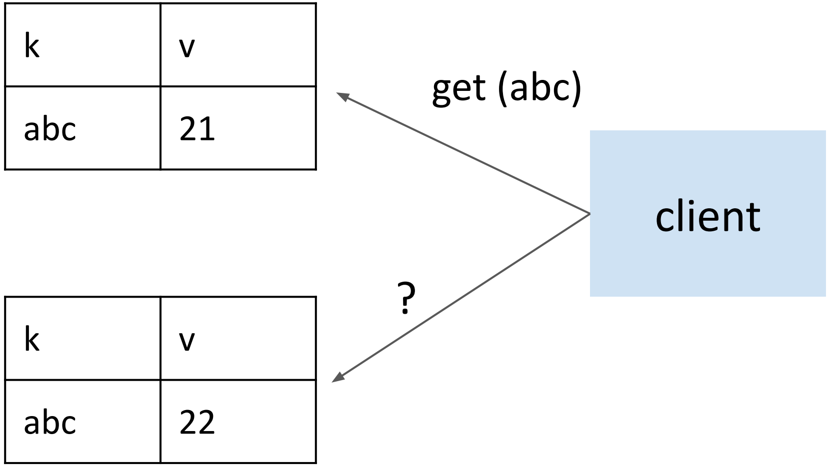
k	v
abc	22

Client 1

Client 2

Example: KV service

- Set (k,v)
- Get (k) -> v



02

Relational Databases

Relational Algebra

$R \cup S$	union
$R \cap S$	intersection
$R \setminus S$	set difference
$R \times S$	Cartesian product
$\pi_{A1, A2, ..., An}(R)$	projection
$\sigma_F(R)$	selection
$R \bowtie S$	natural join
$R \bowtie_\theta S$	theta-join
$R \div S$	division
ρ [A1 B1,.., An Bn]	rename

	Name	Relational Algebra	SQL
Basic / Complete Set	**select**	$\sigma_{age>18}(\text{patients})$	`select * from patients where age > 18`
	project	$\pi_{\text{patient\_id,name}}(\text{patients})$	`select patient_id, name from patients`
	product	$\text{patients} \times \text{medical\_records}$	`select * from patients, medical_records`

	union
	(note: union tables must have the same number of columns and same data types)
		$\pi_{\text{name}}(\text{patients}) \cup \pi_{\text{name}}(\text{doctors})$	`select name from patients`
			`union`
			`select name from doctors`

	difference
	(second table is not necessarily a subset of the first)
		$\pi_{\text{name}}(\text{patients}) - \pi_{\text{name}}(\text{doctors})$	`select name from patients`
			`minus`
			`select name from doctors`

Derived	**rename**	$\rho_{\text{staff}}(\pi_{\text{patient\_id,name}}(\text{patients}))$	`select * from (`
			`select patient_id, name`
			`from patients`
			`) as staff`

	intersection
	note: A intersect B = A-(A-B)
		$\pi_{\text{name}}(\text{patients}) \cap \pi_{\text{name}}(\text{doctors})$	`select name from patients`
			`intersect`
			`select name from doctors`

	natural join note: lecture notes use star, but every other source seems to use bowtie	$\text{patients} \bowtie \text{medical\_records}$ or $\text{patients} * \text{medical\_records}$	`select * from patients` `natural join medical_records`

	theta join
	note: you may sometimes see the '=' replaced with 'θ'
		$\text{patients} \bowtie_{\text{patients.id=doctors.patient\_id}} (\text{doctors})$	`select * from patients`
			`join doctors`
			`on patients.id=doctors.patient_id`

EPIC Institute of Technology
Powered by ‹epam›

SQL Editor | Graphical Query Builder

Previous queries | Delete | Delete All

```sql
SELECT pp.*
FROM (SELECT id, day, start_time, end_time, slot,
             row_number() over (partition by id order by case when current_time
            - interval '1 hours 30 minutes'
              <= end_time
             then 1
                                              else 2
                                         end) as x FROM dish_timings_delivery
      WHERE id in (101999001247, 101999001225, 10199900100, 101999001223,101999001220)
     ) pp
WHERE x < 2;
```

Output pane

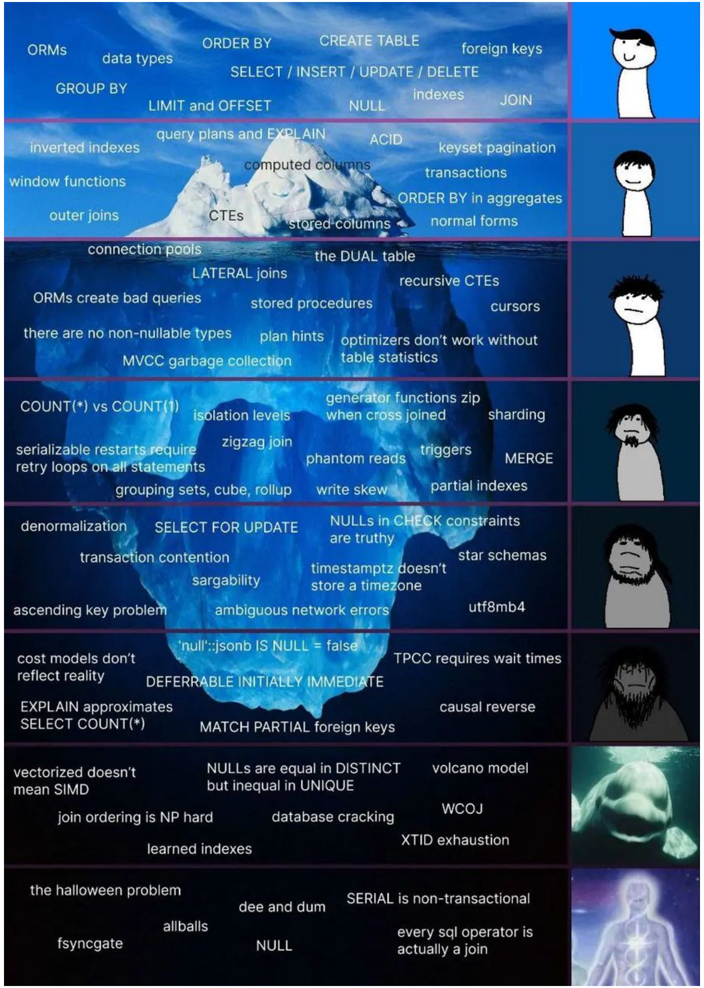
Data Output | Explain | Messages | History

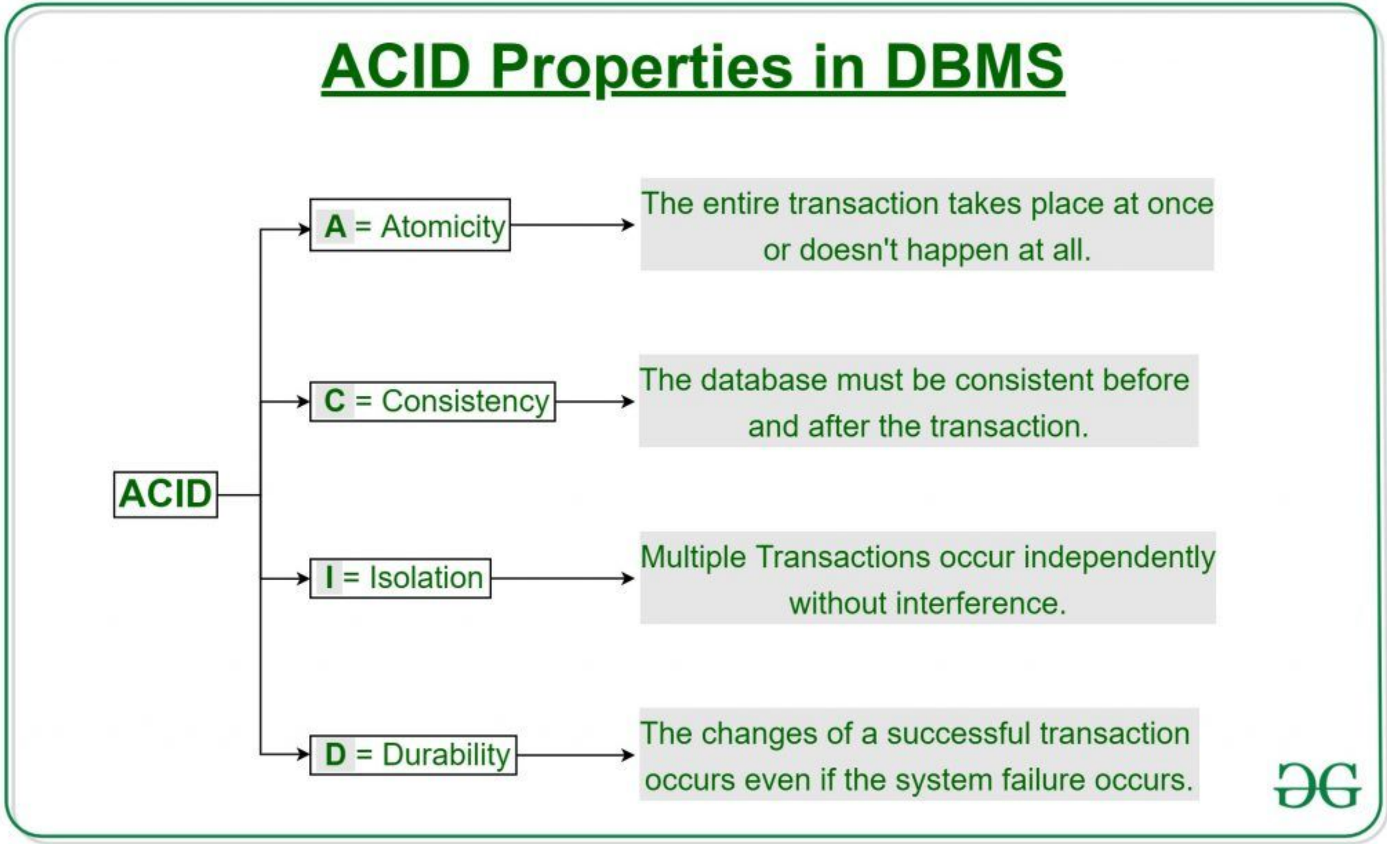
	id bigint	day integer	start_time time without time zone	end_time time without time zone	slot smallint	x bigint
1	10199900100	0	19:00:00	21:00:00	2	1
2	101999001220	0	11:00:00	13:15:00	1	1
3	101999001223	0	19:00:00	20:45:00	2	1
4	101999001225	0	11:00:00	13:15:00	1	1
5	101999001247	0	11:00:00	13:15:00	1	1

OK. Unix Ln 34, Col 86, Ch 1792 5 rows. 92 msec

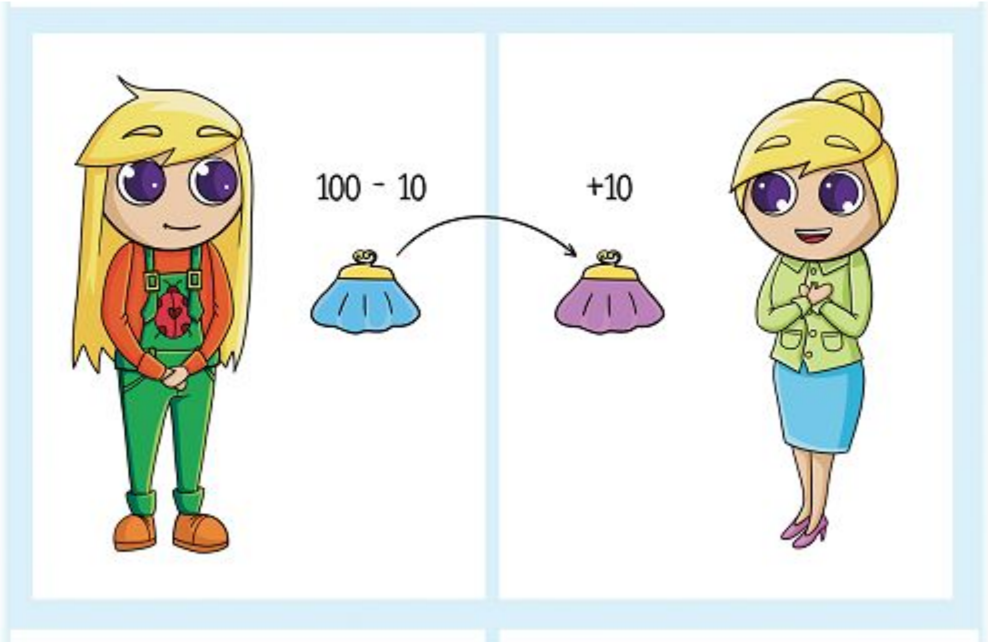
Literal	*Examples*
Character string	'59', 'Python'
Numeric	48, 10.34, 2., .001, -125, +5.33333, 2.5E2, 5E-3
Boolean	TRUE, FALSE, UNKNOWN
Datetime	DATE, '2016-05-14', TIME '04:12:00',TIMESTAMP ‘2016-05-14 10:23:54’
Interval	INTERVAL ‘15-3’ YEAR TO MONTH, INTERVAL ‘23:06:5.5’ HOUR TO SECOND

EPIC Institute of Technology
Powered by ‹epam›

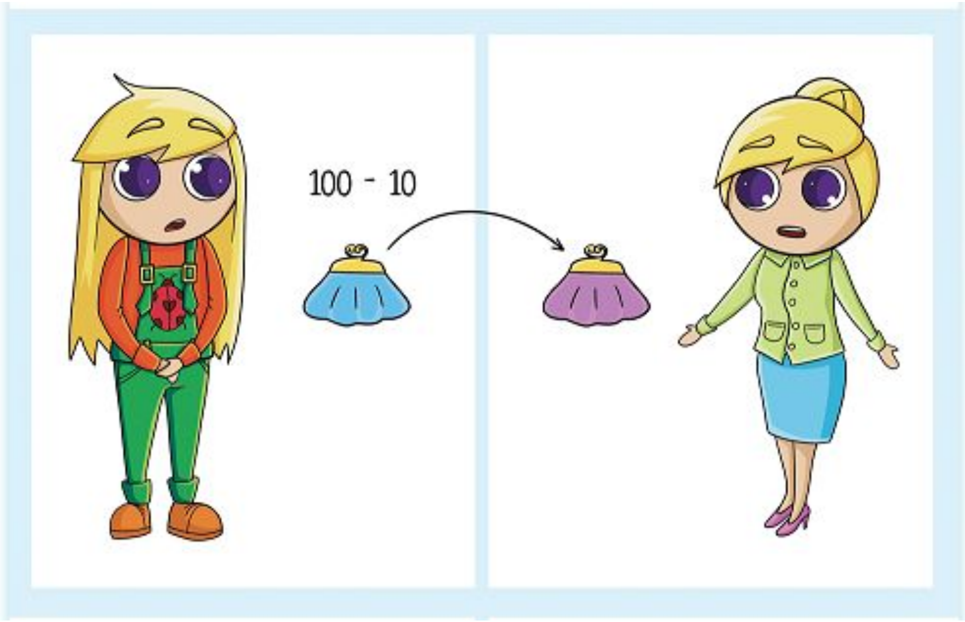



ACID Properties in DBMS
ACID
A = Atomicity
The entire transaction takes place at once or doesn't happen at all.
C = Consistency
The database must be consistent before and after the transaction.
I = Isolation
Multiple Transactions occur independently without interference.
D = Durability
The changes of a successful transaction occurs even if the system failure occurs.

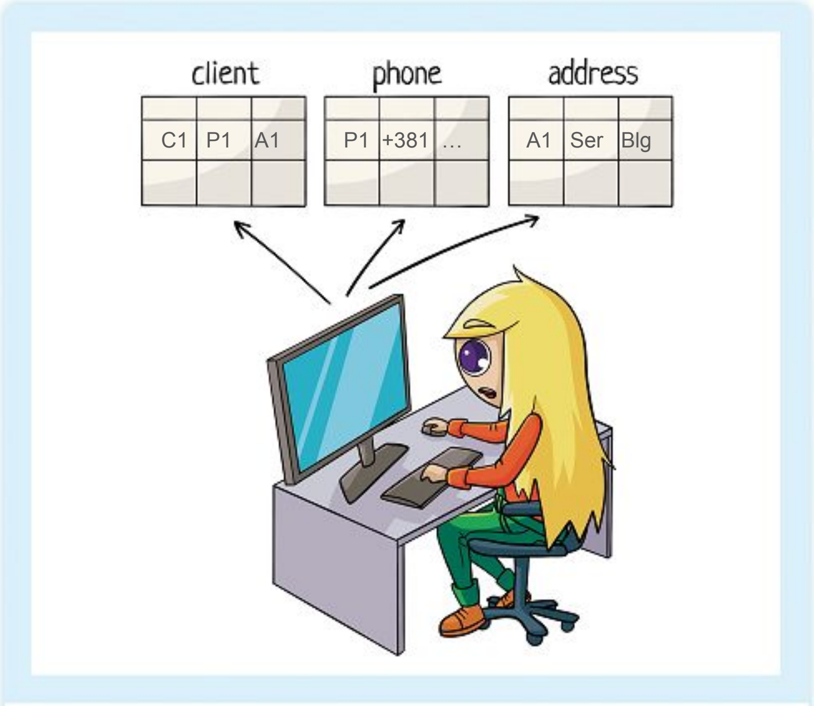
Atomicity



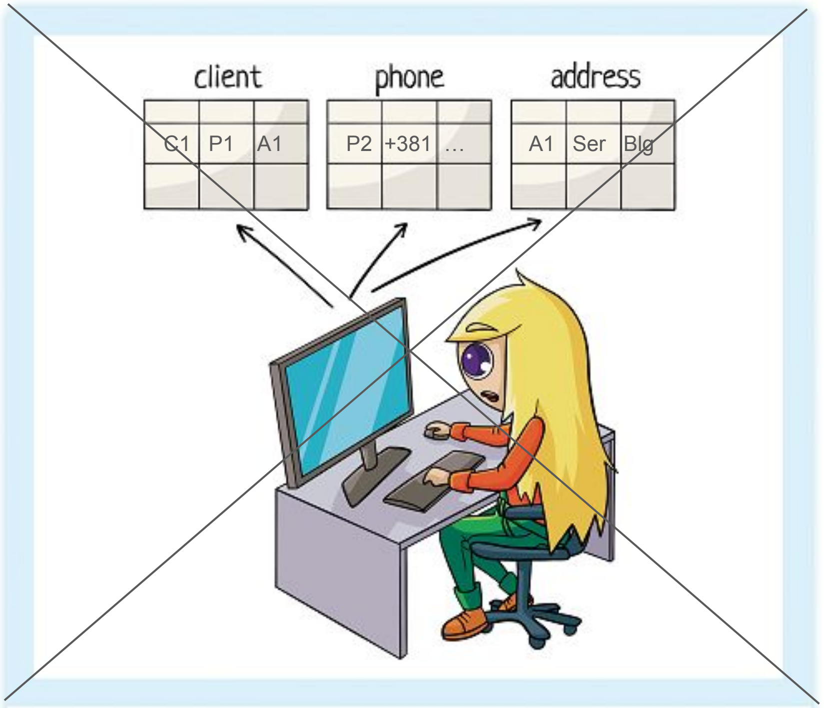
Atomicity



Consistency



Consistency



Isolation

- **Dirty Reads:** Imagine peeking into a room while someone is cleaning. Everything is scattered, and it's a mess. Similarly, a dirty read is like viewing data that another transaction is still changing. Since that change might be reversed, what you're seeing might not stick around.
- **Non-repeatable Reads:** Imagine you note down the number of apples in a basket. A minute later, you count again, but the number has changed because someone took or added an apple. This is what happens here; data you read at the start of a transaction might change by the time the transaction finishes.
- **Phantom Reads:** This is like checking a basket for apples and then finding oranges in your next check. It's unexpected! Phantom reads occur when new data appears (like those mysterious oranges) during a transaction.
- **Lost Updates:** Think of two artists painting on the same canvas. If they paint over each other's work, one of their contributions might disappear. Similarly, when two transactions try to change the same piece of data, one of those changes might get overlooked.

Isolation

Isolation Level	Dirty Read	Non-repeatable reads	Phantom Read	Lost Updates
Read Uncommitted	May Occur	May Occur	May Occur	May Occur
Read Committed	Prevents	May Occur	May Occur	May Occur
Repeatable Read	Prevents	Prevents	May Occur	Prevents
Snapshot	Prevents	Prevents	Prevents	Prevents
Serializable	Prevents	Prevents	Prevents	Prevents

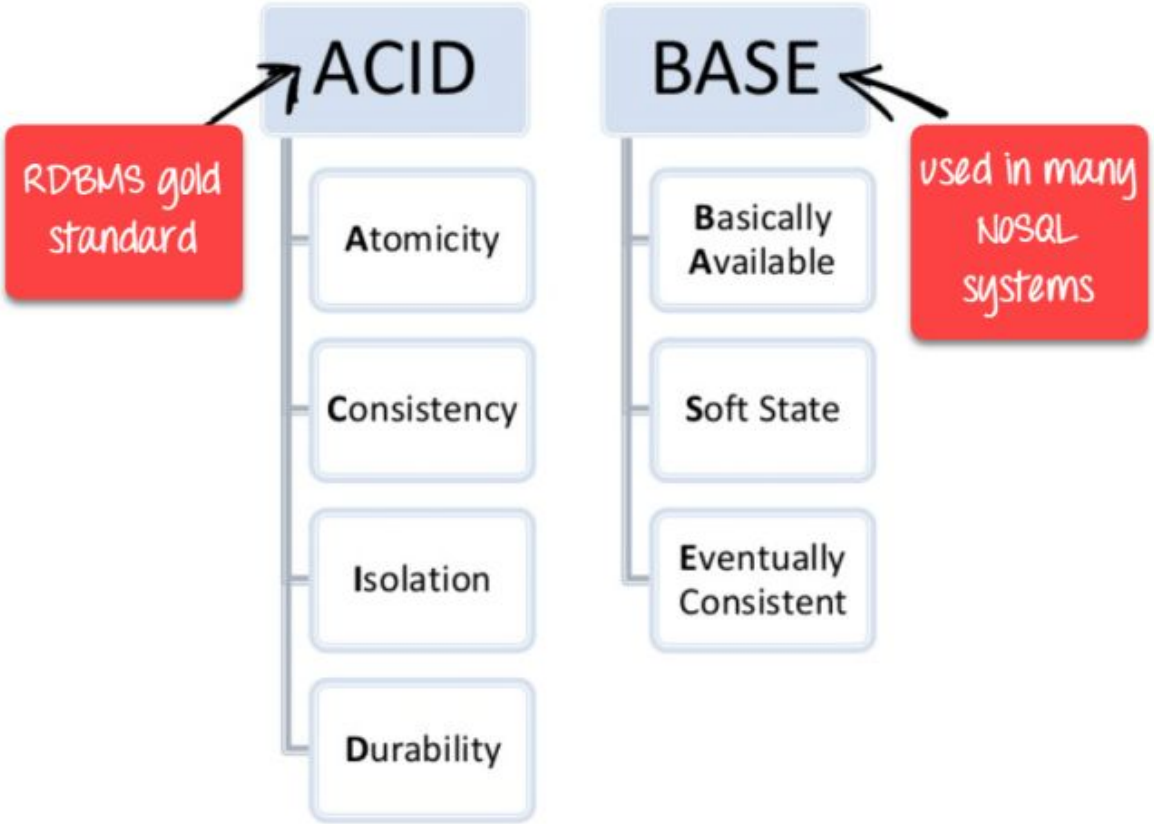
Get isolation

```
simple_bank> show transaction isolation level;
 transaction_isolation
-----------------------
 read committed
(1 row)
```

Change isolation

```
-- Tx1:
simple_bank> begin;
BEGIN

simple_bank> set transaction isolation level read uncommitted;
SET
```


ACID
BASE
RDBMS gold standard
used in many NoSQL systems
Atomicity
Consistency
Isolation
Durability
Basically Available
Soft State
Eventually Consistent

BA

2.a. Basic Availability

As NoSQL prioritises Scalability and Availability over transaction correctness; it needs to be available at all times with highest five 9s percentile (99.999).

S

2.b. Soft State

This is related to eventual consistency. It basically is a disclaimer that the data available in the database is not the final state. Due to eventual consistency across various nodes; the data will not be guaranteed to be *write-consistent* or *mutually consistent* across nodes.

E

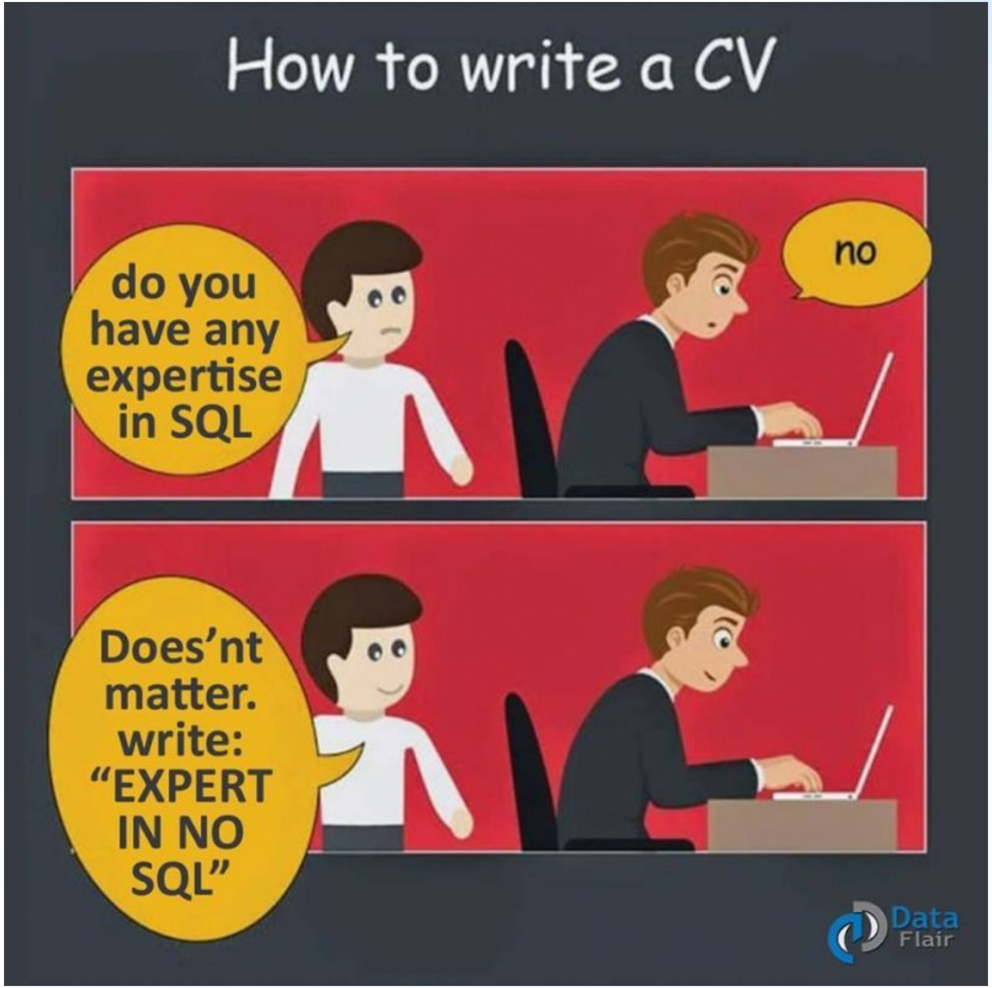
2.c. Eventual Consistency

As there are multiple machines in NoSQL databases due to various reasons such as sharding, horizontal scaling and goal is to be as fast as possible; data might be distributed across various nodes; and whichever node gives the answer first; is treated as the response to the API request. This means that data is consistent eventually across multiple machines. In the initial days, consensus was not the norm in NoSQL databases and hack ways to fan-out requests and pick the first one in order to be first was the norm. Consensus protocol such as Raft and Paxos where then integrated to have consistency (at the expense of performance.)

03

NoSQL

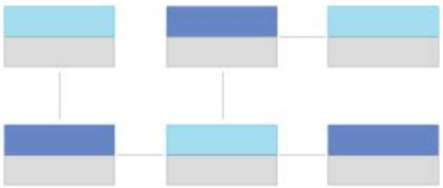
EPIC Institute of Technology
Powered by <epam>



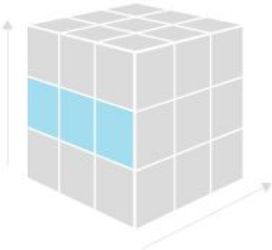
EPIC Institute of Technology
Powered by epam

SQL

Relational

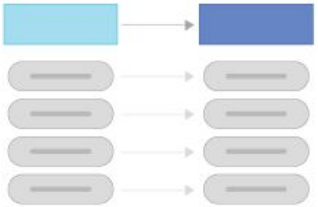


Analytical

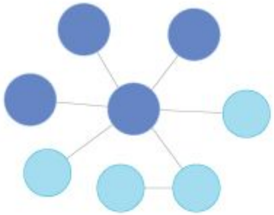


NoSQL

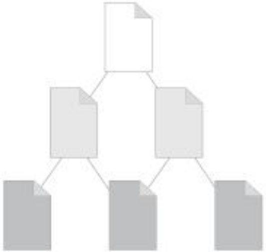
Key - Value



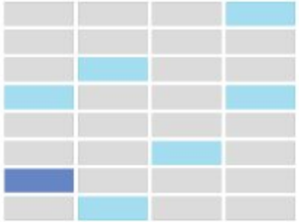
Graph

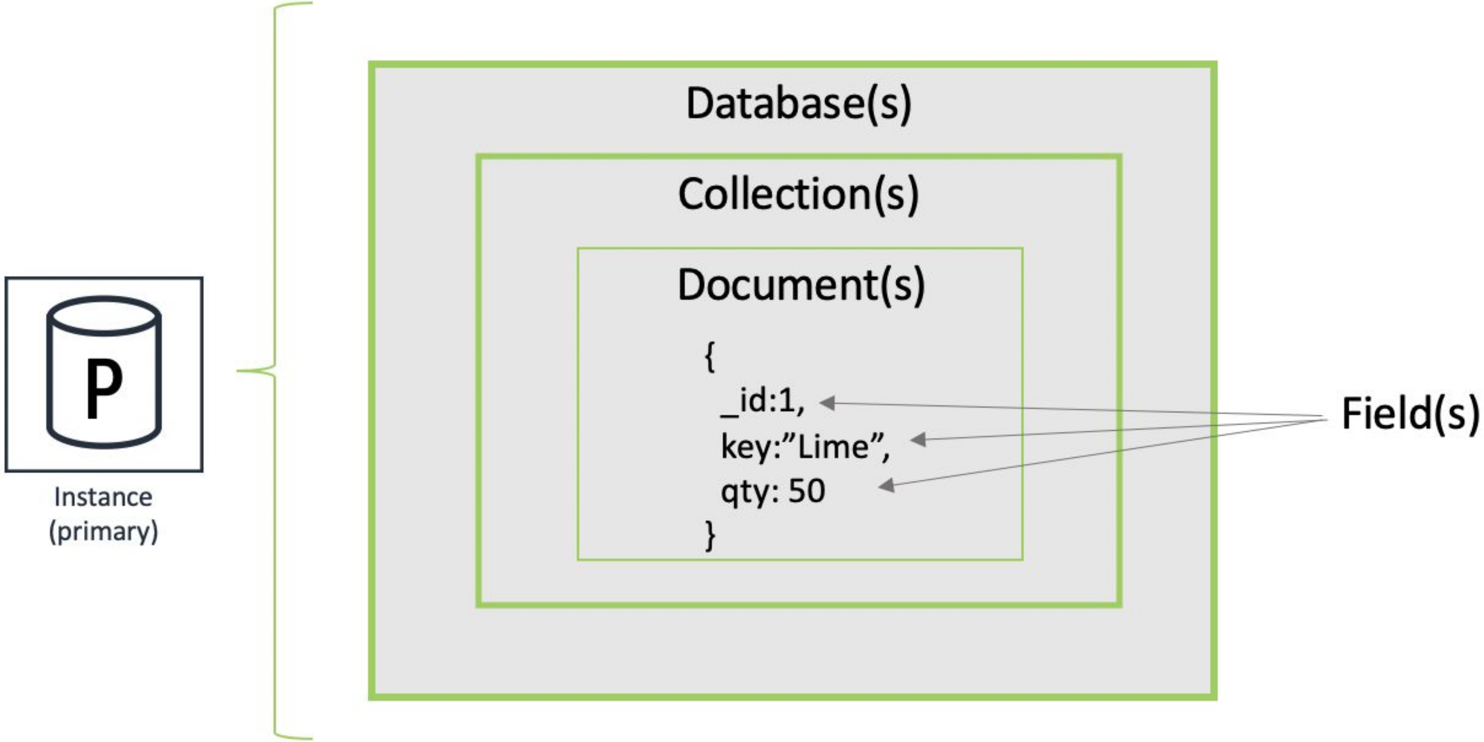


Document



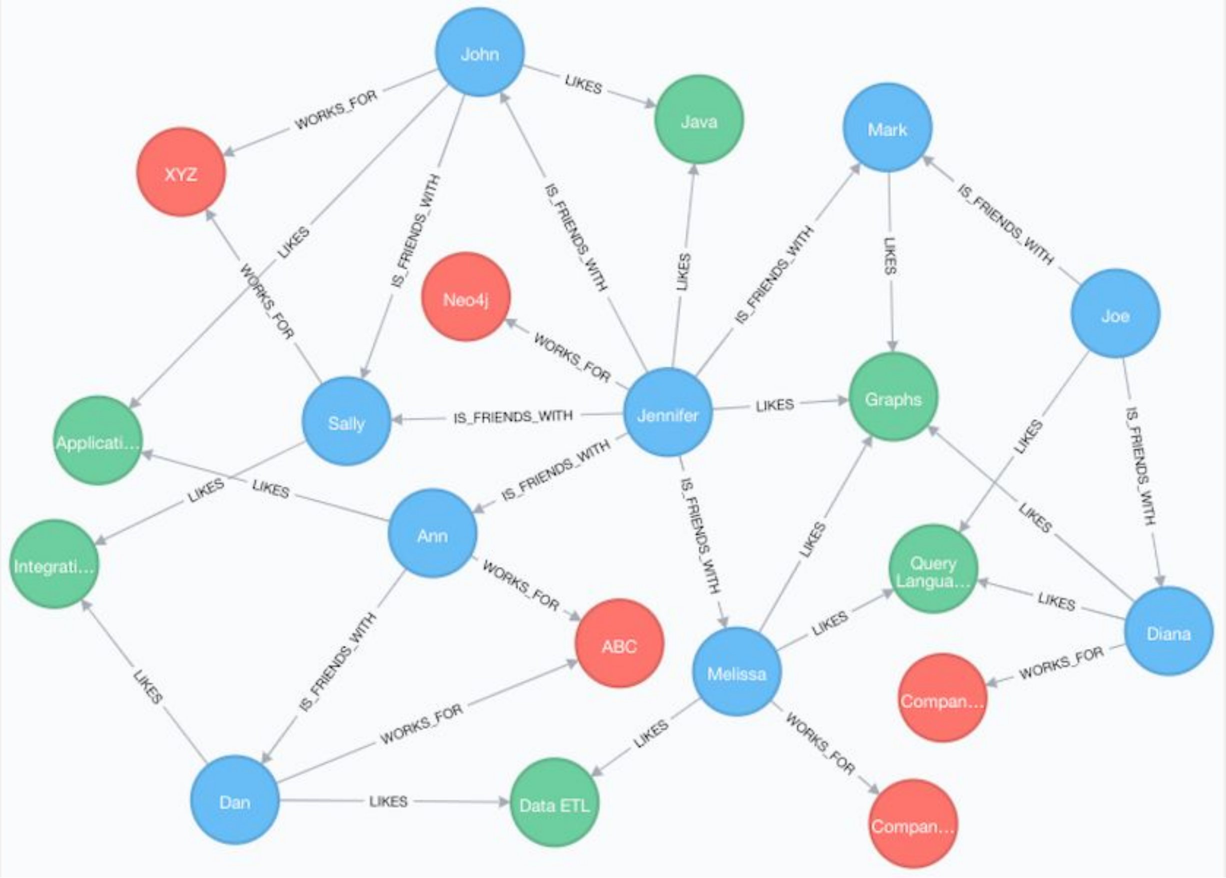
Wide Column





P
Instance
(primary)
Database(s)
Collection(s)
Document(s)
{
_id:1,
key:"Lime",
qty: 50
}
Field(s)

EPIC Institute of Technology
Powered by ‹epam›



EPIC
Institute of Technology
Powered by epam

04

Sharding/replication

Sharding

PRODUCT	PRICE
WIDGET	$118
GIZMO	$88
TRINKET	$37
THINGAMAJIG	$18
DOODAD	$60
TCHOTCHKE	$999

($0-$49.99)

PRODUCT	PRICE
TRINKET	$37
THINGAMAJIG	$18

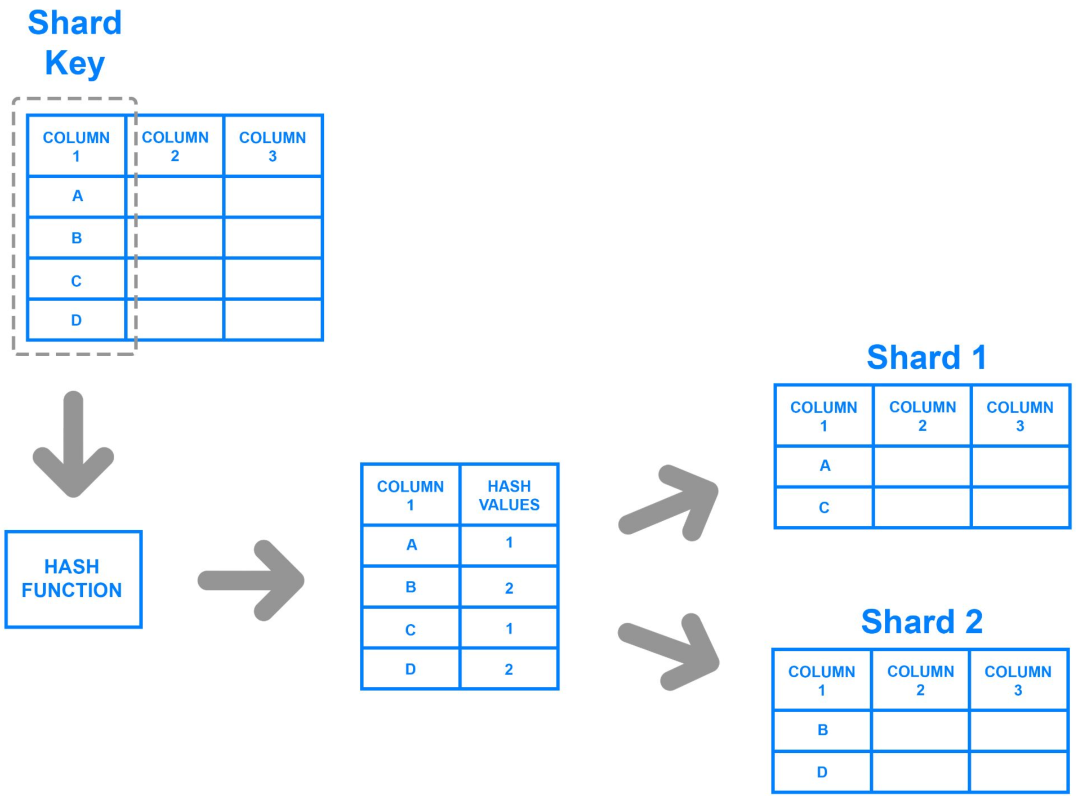
($50-$99.99)

PRODUCT	PRICE
GIZMO	$88
DOODAD	$60

($100+)

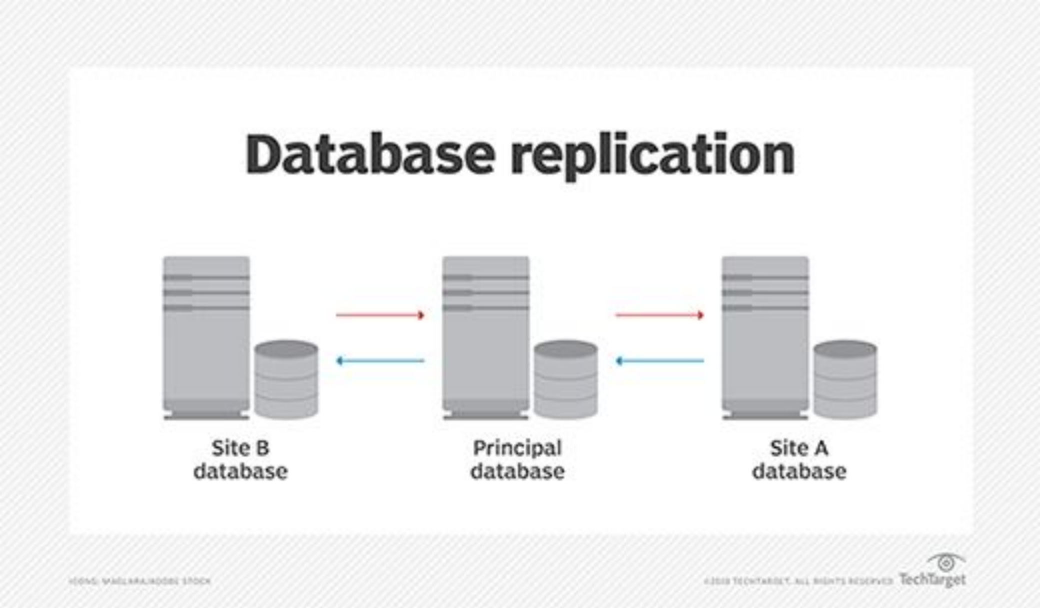
PRODUCT	PRICE
WIDGET	$118
TCHOTCHKE	$999

Sharding



Shard Key
COLUMN 1
COLUMN 2
COLUMN 3
A
B
C
D
HASH FUNCTION
COLUMN 1
HASH VALUES
A
1
B
2
C
1
D
2
Shard 1
COLUMN 1
COLUMN 2
COLUMN 3
A
C
Shard 2
COLUMN 1
COLUMN 2
COLUMN 3
B
D

Replication



Goals

- Performance -> Sharding
    - Split a big file in small parts (shards, chunks), store them on different machines
- Fault tolerance -> Replication
    - Duplicate chunks so data is not lost ● Replication -> (IN) Consistency
    - How do we guarantee that replicas are the same? ● Consistency -> Performance ?
    - Making the system more consistent decreases the speedup we gain from sharding

05

Indexes

Goals

Index

friends_name_asc	
name	**id**
Andrew	3
Blake	5
Dave	2
Evan	6
Matt	1
Nick	7
Todd	4
Zack	8

points to

Table

Friends		
id	**name**	**city**
1	Matt	San Francisco
2	Dave	Oakland
3	Andrew	Blacksburg
4	Todd	Chicago
5	Blake	Atlanta
6	Evan	Detroit
7	Nick	New York City
8	Zack	Seattle

EPIC Institute of Technology
Powered by <epam>

Structures

- **Comparison** (B-tree)
- **Space indexes** (R-Tree/Grid-based Spatial/Quadtree)
- Equality (Hash table)
- Other (Bitmap, Reverse, Inverted, Partial, Function-based)

11.2. Index Types

11.2.1. B-Tree
11.2.2. Hash
11.2.3. GiST
11.2.4. SP-GiST
11.2.5. GIN
11.2.6. BRIN

EPIC Institute of Technology
Powered by <epam>

B-tree

11.2.1. B-Tree

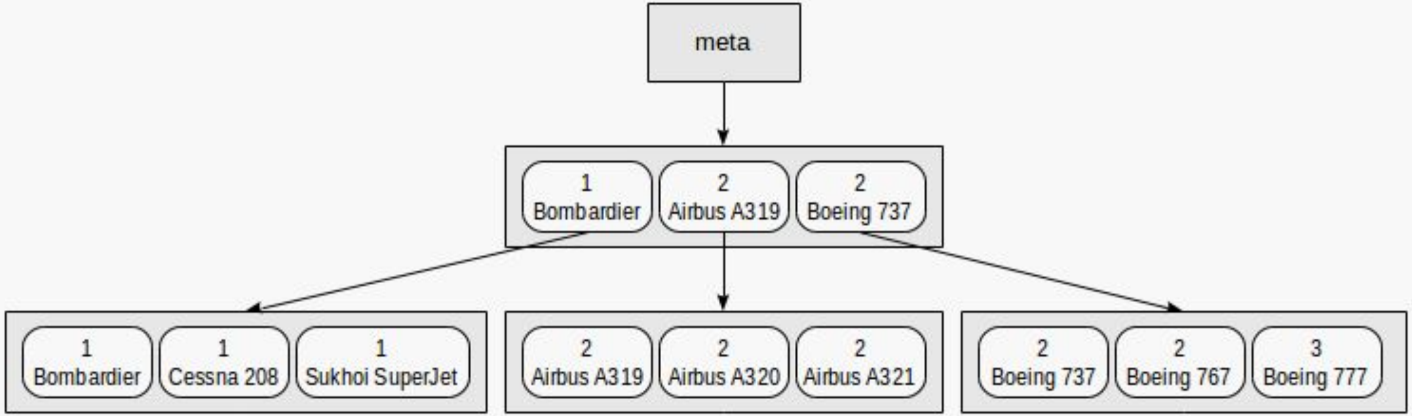
B-trees can handle equality and range queries on data that can be sorted into some ordering. In particular, the PostgreSQL query planner will consider using a B-tree index whenever an indexed column is involved in a comparison using one of these operators:

```
<   <=   =   >=   >
```

Constructs equivalent to combinations of these operators, such as `BETWEEN` and `IN`, can also be implemented with a B-tree index search. Also, an `IS NULL` or `IS NOT NULL` condition on an index column can be used with a B-tree index.

EPIC
Institute of Technology
Powered by ‹epam›

B-tree



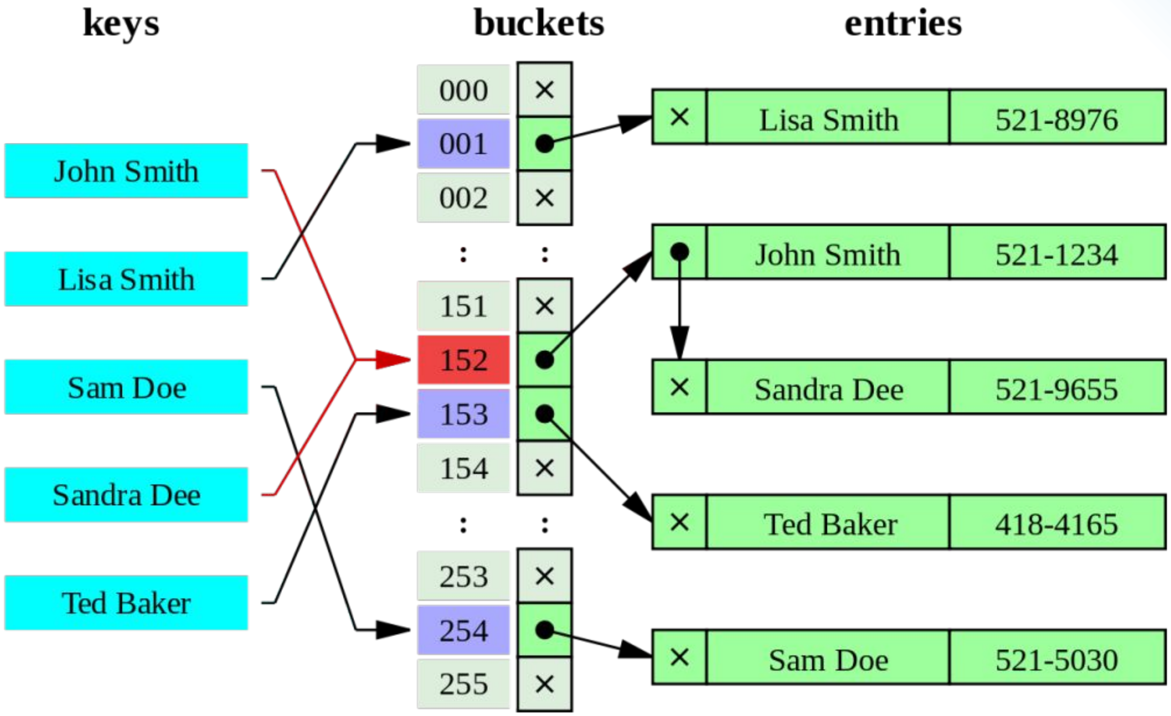
Hash

11.2.2. Hash

Hash indexes store a 32-bit hash code derived from the value of the indexed column. Hence, such indexes can only handle simple equality comparisons. The query planner will consider using a hash index whenever an indexed column is involved in a comparison using the equal operator:

```
=
```

Hash


keys
buckets
entries
John Smith
Lisa Smith
Sam Doe
Sandra Dee
Ted Baker
000
001
002
:
151
152
153
154
:
253
254
255
Lisa Smith
521-8976
John Smith
521-1234
Sandra Dee
521-9655
Ted Baker
418-4165
Sam Doe
521-5030

EPIC Institute of Technology
Powered by <epam>

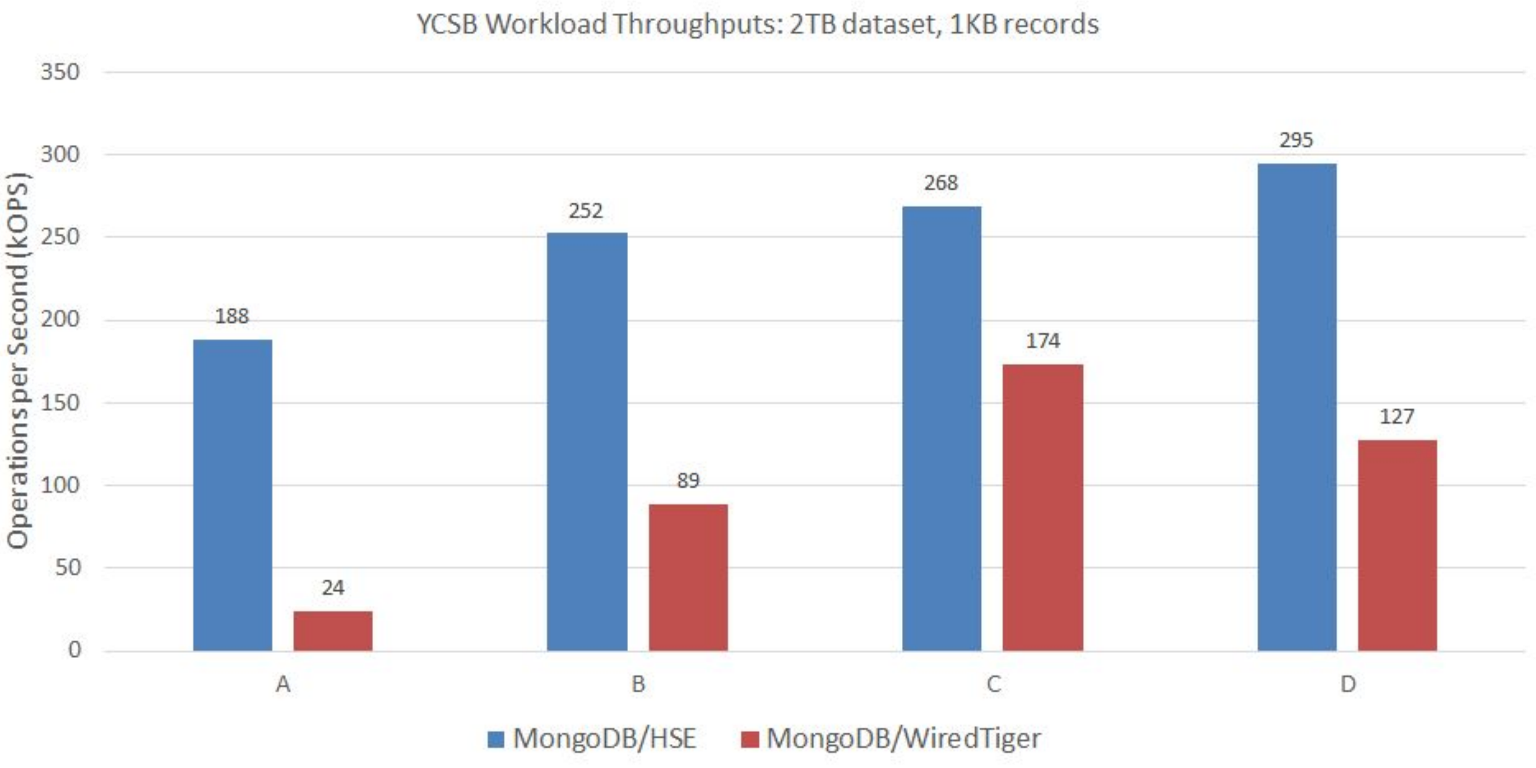
06

Comparison

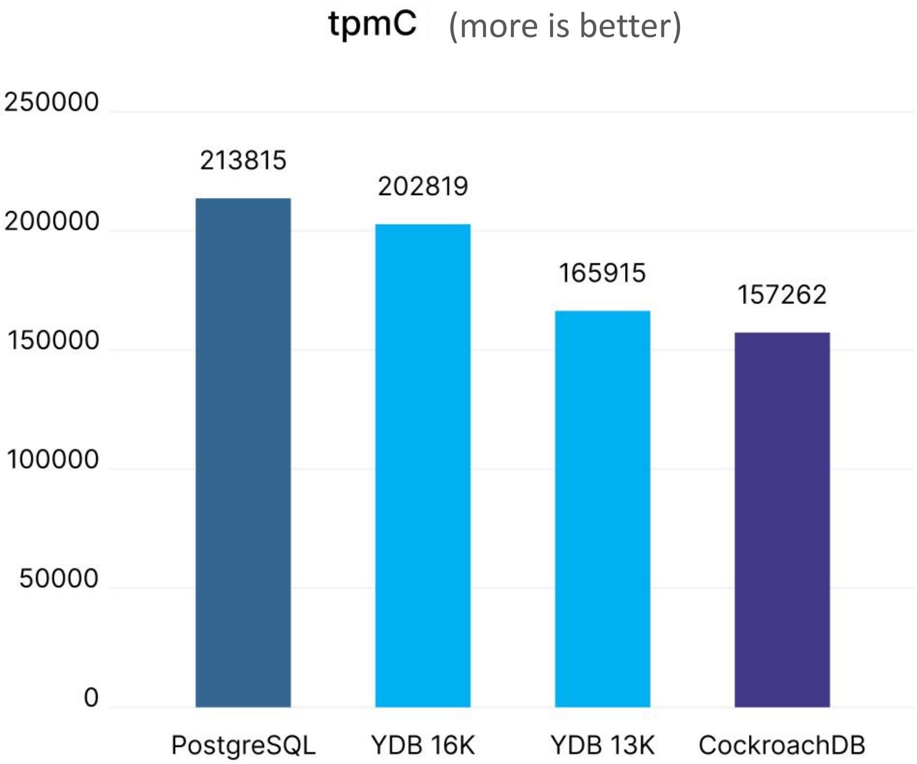
	MongoDB	PostgreSQL
Data Model	Document-based (NoSQL)	Relational (SQL)
Data Types	Semi-structured and unstructured data	Structured data
Schema Flexibility	Dynamic, flexible schemas	Rigid, predefined schemas
Query Language	MQL	SQL
Query Capabilities	Simple queries and aggregation pipelines	Complex queries and analytical pipelines
Scalability	Horizontally scalable (sharding)	Vertical scaling
ACID Transactions	Document-level ACID transaction support	Fully ACID compliant

Comparison of NoSQL databases

DATABASE	TYPE	VENDOR OR OPEN SOURCE	ACID COMPLIANCE	PRIMARY QUERY LANGUAGE	TOP USE CASES	SECURITY
Couchbase	Document-based, key value	Open source	Yes	N1QL	Customer service, financial services, inventory and IoT	Includes security for authentication, encryption, auditing and authorization
Cassandra	Wide column	Open source	No	CQL	Social analytics, real-time analytics, retail and messaging	Built-in security for authorization, encryption and authentication, but security is disabled by default for ease of use within clusters
Neo4j	Graph	Open source single-node version; commercial license for clustering	Yes	Cypher	AI, master data management, recommendation services and fraud protection	Built-in security for authorization, roles and encryption
Google Cloud Bigtable	Wide column	Vendor	No	Allows for use of many languages	IoT data management, financial services, retail data and time series data	Secured by vendor
Redis	Key value	Open source	Yes	Allows for use of many languages	Caching, queuing, filtering and stats	Automatically starts in "protection mode" and offers security suggestions
MongoDB	Document-based	Limited open source version; advanced features require commercial subscription	Yes	JavaScript	IoT management, real-time analytics, app development, inventory and personalization	Built-in security for authorization, authentication and encryption
Amazon DynamoDB	Key value or document-based	Vendor	Yes	DQL	Gaming, retail, financial services, advertising and streaming media	Built-in security for data and applications; vendor-secured software, hardware, facilities and network


YCSB Workload Throughputs: 2TB dataset, 1KB records
Operations per Second (kOPS)
350
300
250
200
150
100
50
0
188
24
252
89
268
174
295
127
A
B
C
D
MongoDB/HSE
MongoDB/WiredTiger

EPIC Institute of Technology
Powered by epam



EPIC Institute of Technology
Powered by epam

RANK	DATABASE	CLOUD	THROUGHPUT [ops/s]	READ LATENCY [ms]	WRITE LATENCY [ms]	MONTHLY COSTS [$]	THROUGHPUT PER COST [ops/s/$]
1	ScyllaDB v4.5.1 OpenSource vanilla large	AWS large	**204 405**	4,9	5,4	3 089	66,20
2	Cassandra Apache v4.0.0 OpenSource vanilla large	Alibaba Cloud large	**196 364**	6,3	6,0	2 877	68,25
3	Cassandra Apache v4.0.0 Community vanilla large	IONOS Cloud large	**166 018**	4,2	4,0	4 112	40,40
4	Couchbase Server CE v7.0.0 Community vanilla	Alibaba Cloud large	**153 519**	1,8	1,8	959	160,00

[source]

07

GFS

[Google File System (2003) Sanjay Ghemawat]

Goals / Design

- Store big data (web crawls, reverse indexes)
- Work on usual hardware
- Operate under assumption that failures (many) can happen
- Big files
- Sequential reads / writes
- Append rather than overwrite
- Practical

Overall design

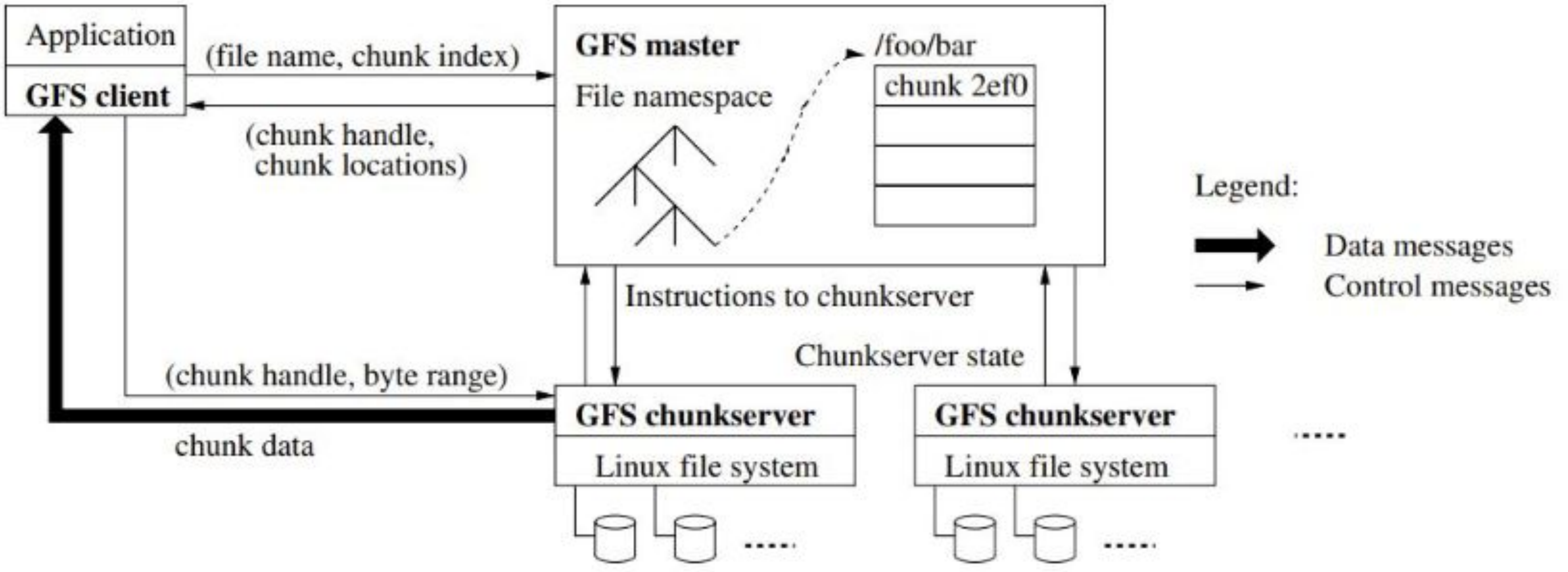


Figure 1: GFS Architecture

Overall design

- “Single” master, multiple chunk servers
- Master
    - Filename -> [chunk handlers]
    - Handle -> [chunckserver id, version, primary, lease expiration]
    - Log (+ checkpoints)
    - Use RAM for efficiency + disk for robustness for fault tolerance

Challenges

- Master as a single weak spot
    - Shadow master
    - Minimize master usage
- How to restore state in case of failure?
    - We log all operations and consider operation successful only when we write on disk
    - To reduce the recovery time we store checkpoints (current state at a given point in time)
- ...

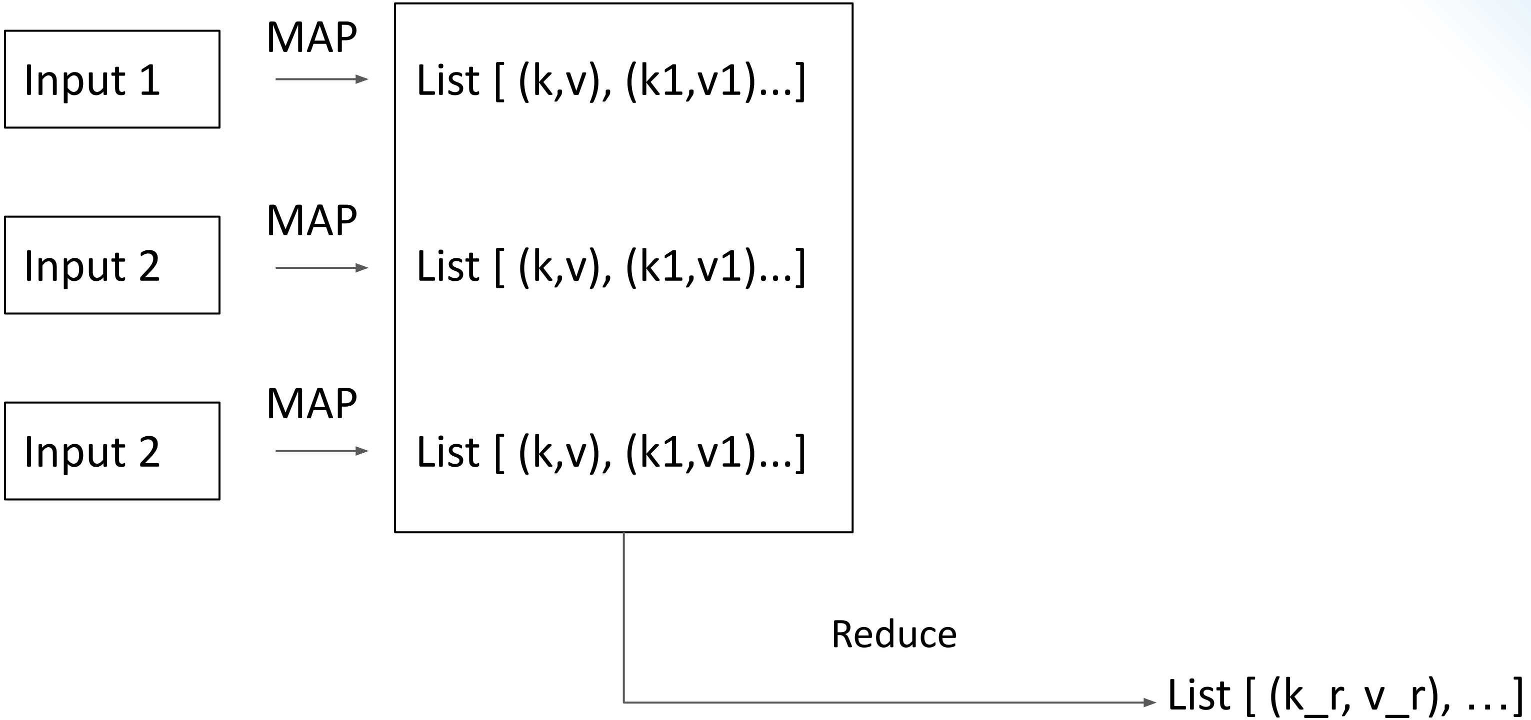
Reads

- Client [filename, offset] -> Master
- Return list of chunkservers, cache them
- Client contacts chunkservers from now on

Writes

- Is there primary?
    - We need version to figure out where up-to-date replicas are located
    - Master remembers the last version
- Primary is responsible for ordering the operations, secondaries follow
- Success - primary and all secondaries have the successful write operation executed

Mapreduce


Input 1
MAP
List [(k,v), (k1,v1)...]
Input 2
MAP
List [(k,v), (k1,v1)...]
Input 2
MAP
List [(k,v), (k1,v1)...]
Reduce
List [(k_r, v_r), ...]

Reverse index

MAP

Input: website

Split to words, emit (word, URL)

Reduce

INPUT: k: word, v: iterator

Emit (word, list (websites))

Search: intersect the list of websites for all words in query

Another task

```
function Map is
    input: integer K1 between 1 and 1100, representing a batch of 1 million social.person records
    for each social.person record in the K1 batch do
        let Y be the person's age
        let N be the number of contacts the person has
        produce one output record (Y,(N,1))
    repeat
end function

function Reduce is
    input: age (in years) Y
    for each input record (Y,(N,C)) do
        Accumulate in S the sum of N*C
        Accumulate in Cnew the sum of C
    repeat
    let A be S/Cnew
    produce one output record (Y,(A,Cnew))
end function
```

Example(with count)

Data - {14, 2}, {14, 1}, {14, 5}, {14, 6}

Mapped - {14, 2, 1}, {14, 1, 1}, {14, 5, 1}, {14, 6, 1}

Reduced(1 machine) - {14, 14, 4} (answer - 14 / 4)

Reduced(2 machines) - {14, 3, 2}, {14, 11, 2} (answer - 14 / 4)

Example(wo count)

Data - {14, 2}, {14, 1}, {14, 5}, {14, 6}

Mapped - {14, 2}, {14, 1}, {14, 5}, {14, 6}

Reduced(1 machine) - {14, 14 / 4} (answer - 14 / 4 = 3.5)

Reduced(2 machines) - {14, 3 / 2}, {14, 11 / 2} (answer = 7)

Now

APACHE
hadoop™

Example

```
select fruit, sum(price)
from log
where event = 'buy'
group by fruit;
```

EPIC Institute of Technology
Powered by ‹epam›

Data

user	event	fruit	weight	price	time
petrovich	buy	banana	2000	105	12:00
isolda_lvovna	buy	apple	150	20	13:15
isolda_lvovna	buy	peach	300	70	13:20

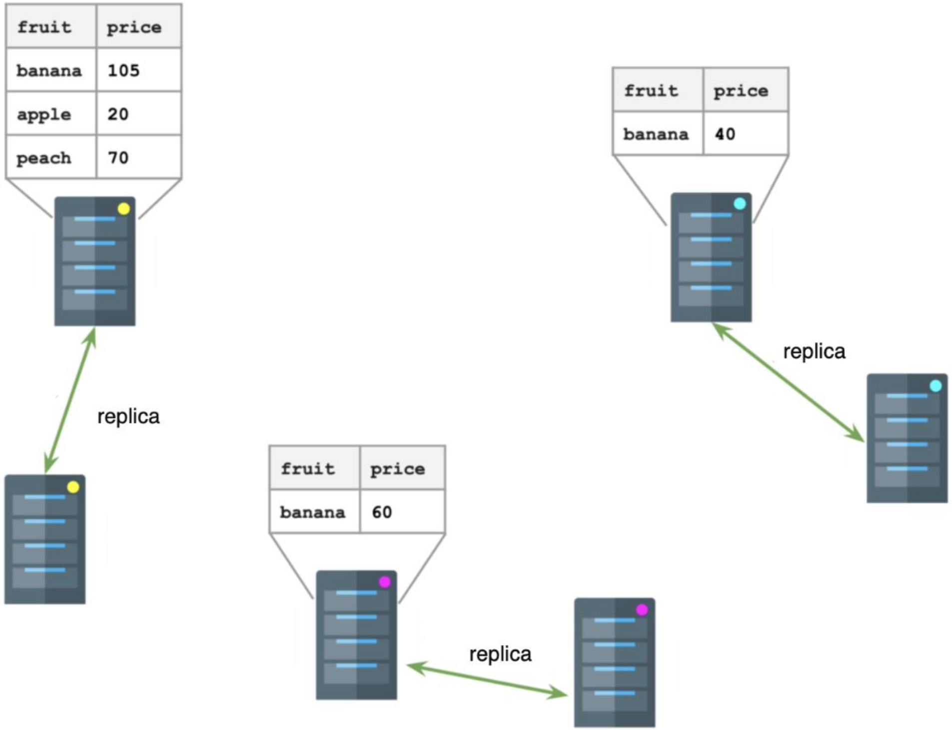
user	event	fruit	weight	price	time
isolda_lvovna	buy	banana	200	40	13:30
ashot	sell	apple	10000	600	14:00
ashot	sell	banana	10000	400	14:10

user	event	fruit	weight	price	time
sonechka	test	apple	50	0	15:23
sonechka	test	apple	50	0	15:24
sonechka	buy	banana	300	60	15:30

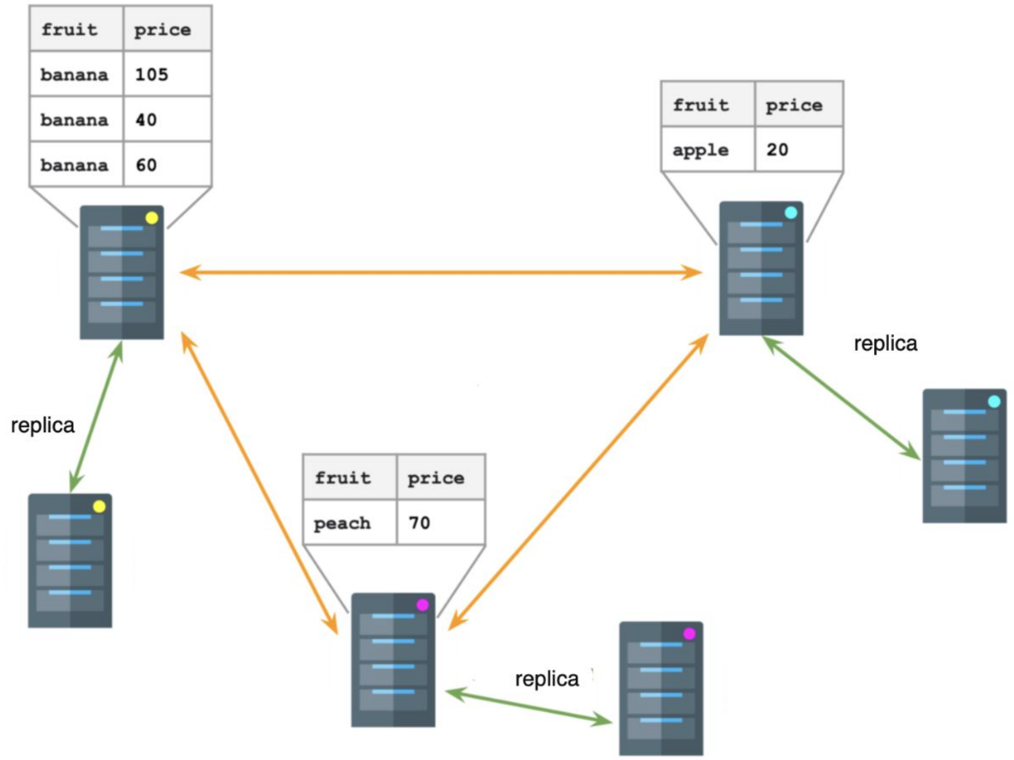
Map function

```
void map(Row row, Context context) {
    if (row.get("event").equals("buy")) {
        context.write(new Row(
            "fruit", row.get("fruit"),
        "price", row.get("price")
        ));
    }
}
```

Map result


fruit price
banana 105
apple 20
peach 70
replica
fruit price
banana 40
replica
fruit price
banana 60
replica

Shuffle



fruit
price
banana
105
banana
40
banana
60
fruit
price
apple
20
replica
replica
fruit
price
peach
70
replica

Reduce func

```
void reduce(List<Row> rows, Context context) {
    int sum = 0;
    for (Row row : rows) {
        sum += row.get("price");
    }
    context.write(new Row(
        "fruit", rows.get(0).get("fruit"),
        "price", sum
    ));
}
```

Reduce

fruit
price
banana
205
fruit
price
apple
20
fruit
price
peach
70