

NoSQL

Theory, Implementations, an introduction

NoSQL

- What does it mean?
 - ▣ Not Only SQL.

Use Cases

- ❑ Massive write performance.
- ❑ Fast key value look ups.
- ❑ Flexible schema and data types.
- ❑ No single point of failure.
- ❑ Fast prototyping and development.
- ❑ Out of the box scalability.
- ❑ Easy maintenance.

Motives Behind NoSQL

- Big data.
- Scalability.
- Data format.
- Manageability.

Big Data

- Collect.
- Store.
- Organize.
- Analyze.
- Share.

Data growth outruns the ability to manage it so we need **scalable** solutions.

Scalability

- Scale up, Vertical scalability.
 - ▣ Increasing server capacity.
 - ▣ Adding more CPU, RAM.
 - ▣ Managing is hard.
 - ▣ Possible down times

Scalability

- Scale out, Horizontal scalability.
 - ▣ Adding servers to existing system with little effort, aka Elastically scalable.
 - Bugs, hardware errors, things fail all the time.
 - It should become cheaper. Cost efficiency.
 - ▣ Shared nothing.
 - ▣ Use of commodity/cheap hardware.
 - ▣ Heterogeneous systems.
 - ▣ Controlled Concurrency (avoid locks).
 - ▣ Service Oriented Architecture. Local states.
 - Decentralized to reduce bottlenecks.
 - Avoid Single point of failures.
 - ▣ Asynchrony.
 - ▣ Symmetry, you don't have to know what is happening. All nodes should be symmetric.

What is Wrong With RDBMS?

- ❑ Nothing. One size fits all? Not really.
- ❑ Impedance mismatch.
 - ❑ Object Relational Mapping doesn't work quite well.
- ❑ Rigid schema design.
- ❑ Harder to scale.
- ❑ Replication.
- ❑ Joins across multiple nodes? Hard.
- ❑ How does RDMS handle data growth? Hard.
- ❑ Need for a DBA.
- ❑ Many programmers are already familiar with it.
- ❑ Transactions and ACID make development easy.
- ❑ Lots of tools to use.

ACID Semantics

- Atomicity: All or nothing.
- Consistency: Consistent state of data and transactions.
- Isolation: Transactions are isolated from each other.
- Durability: When the transaction is committed, state will be durable.

Any data store can achieve Atomicity, Isolation and Durability but do you always need consistency? No.

By giving up ACID properties, one can achieve higher performance and scalability.

Enter CAP Theorem

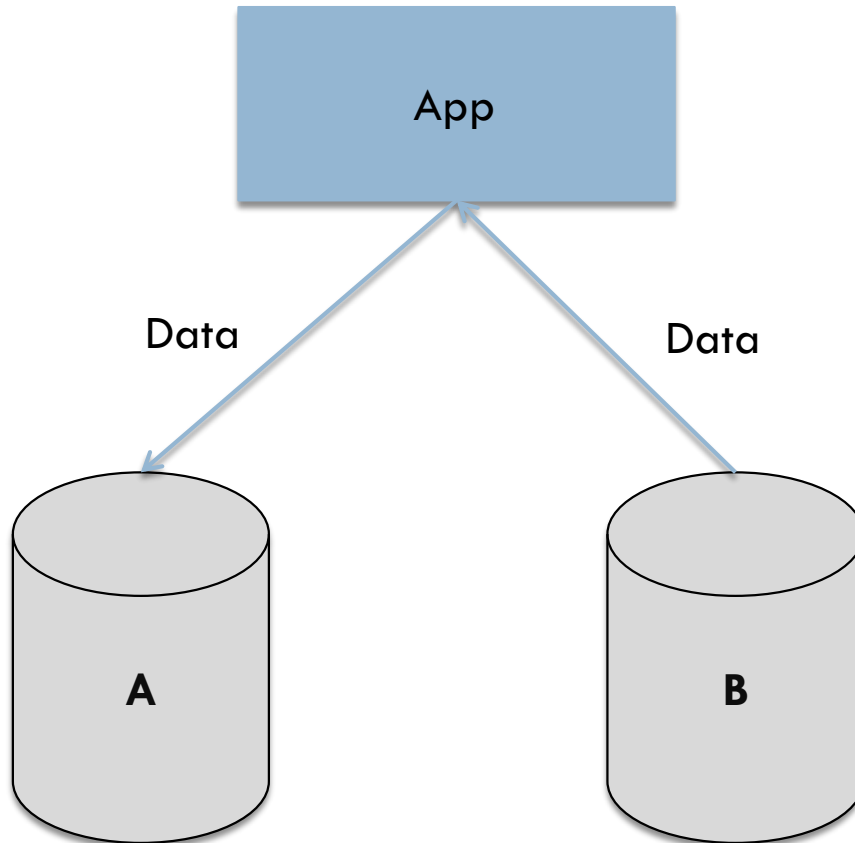
- Also known as Brewer's Theorem by Prof. Eric Brewer, published in 2000 at University of Berkeley.
- “Of three properties of a shared data system: data consistency, system availability and tolerance to network partitions, only two can be achieved **at any given moment.**”
- Proven by Nancy Lynch et al. MIT labs.

CAP Semantics

- **Consistency:** Clients should read the same data. There are many levels of consistency.
 - ▣ Strict Consistency – RDBMS.
 - ▣ Tunable Consistency – Cassandra.
 - ▣ Eventual Consistency – Amazon Dynamo.
- **Availability:** Data to be available.
- **Partial Tolerance:** Data to be partitioned across network segments due to network failures.

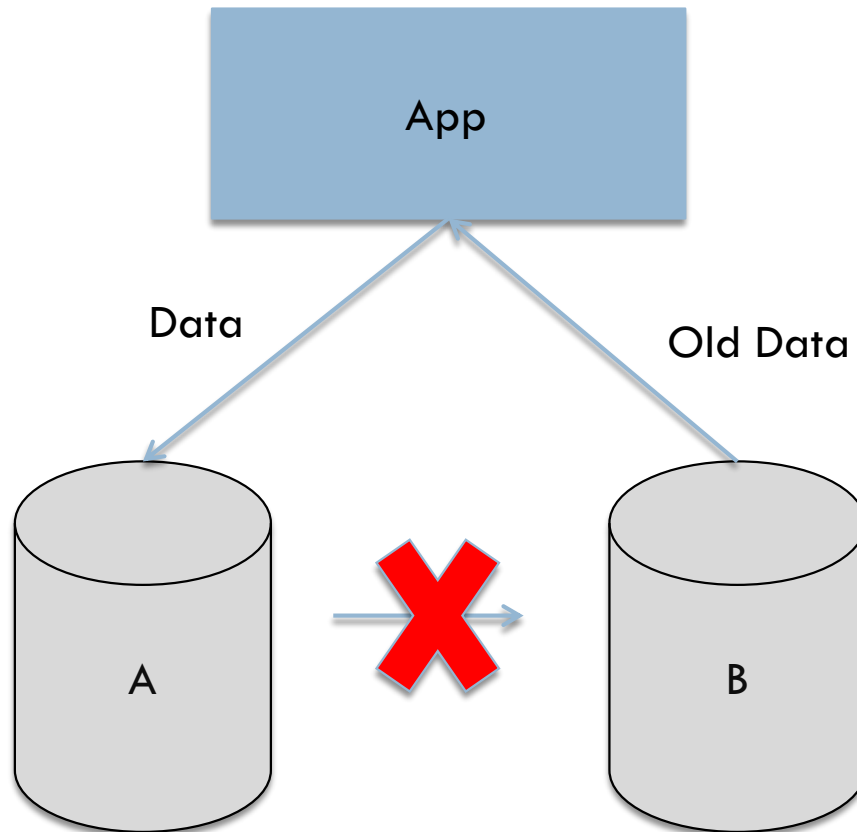
A Simple Proof

Consistent and available
No partition.



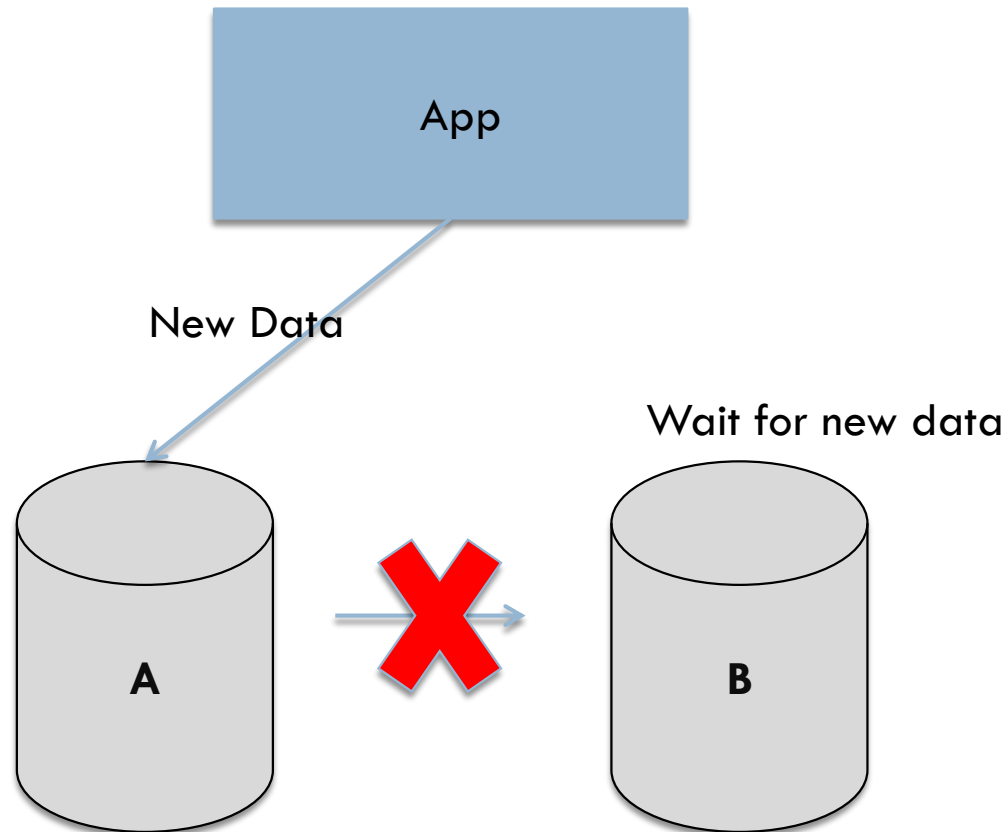
A Simple Proof

Available and partitioned
Not consistent, we get back old data.



A Simple Proof

Consistent and partitioned
Not available, waiting...



BASE, an ACID Alternative

Almost the opposite of ACID.

- Basically available: Nodes in the a distributed environment can go down, but the whole system shouldn't be affected.
- Soft State (scalable): The state of the system and data changes over time.
- Eventual Consistency: Given enough time, data will be consistent across the distributed system.

A Clash of cultures

ACID:

- Strong consistency.
- Less availability.
- Pessimistic concurrency.
- Complex.

BASE:

- Availability is the most important thing. Willing to sacrifice for this (CAP).
- Weaker consistency (Eventual).
- Best effort.
- Simple and fast.
- Optimistic.

Distributed Transactions

□ Two phase commit.

- Starbucks doesn't use two phase commit by Gregor Hoppe.

□ Possible failures

- Network errors.
- Node errors.
- Database errors.

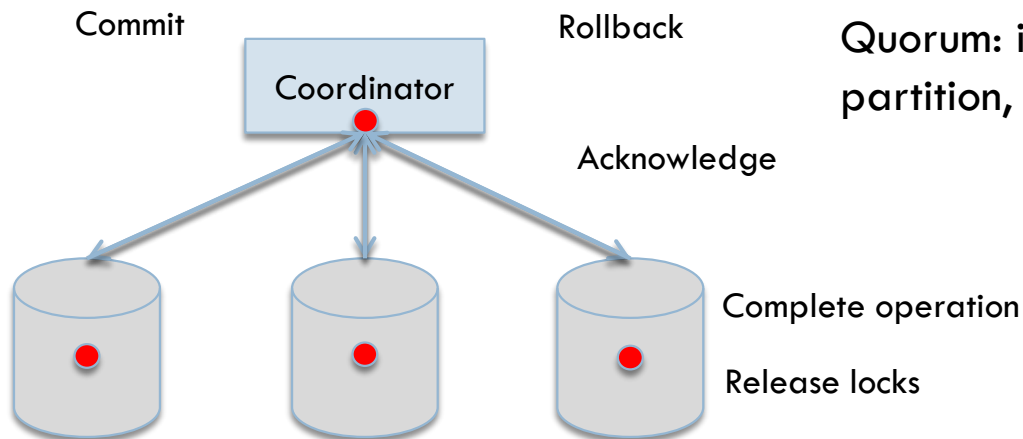
Problems:

Locking the entire cluster if one node is down

Possible to implement timeouts.

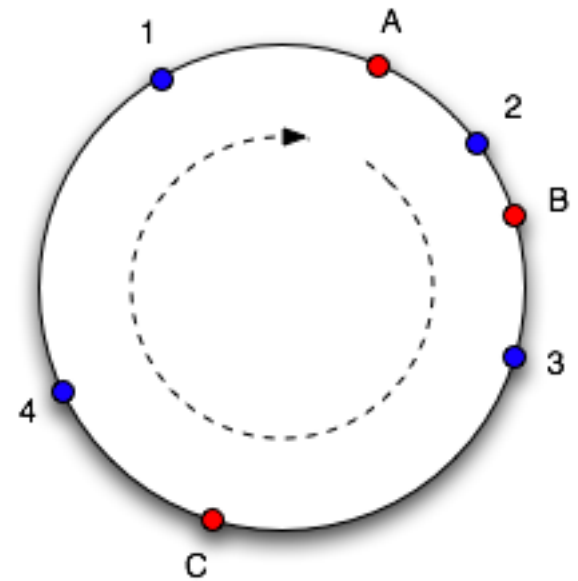
Possible to use Quorum.

Quorum: in a distributed environment, if there is partition, then the nodes vote to commit or rollback.



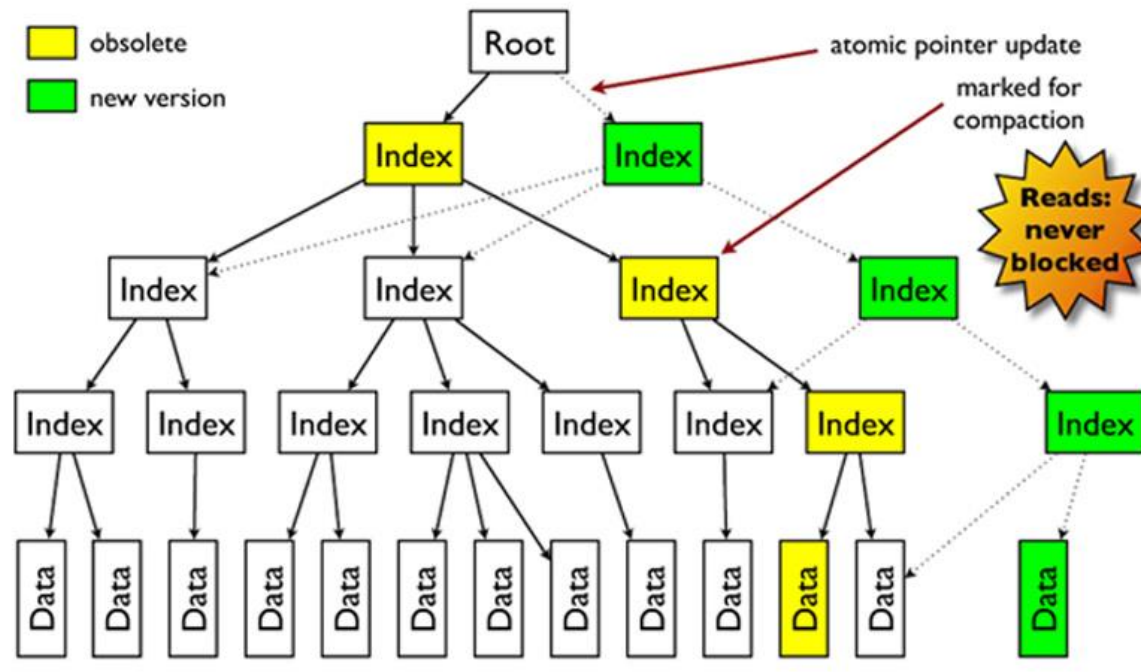
Consistent Hashing

- Solves Partitioning Problem.
- Consistent Hashing, Memcached.
 - ▣ `servers = [s1, s2, s3, s4, s5]`
 - ▣ `serverToSendData = servers[hash(data) % servers.length]`
- A New Hope
 - ▣ Continuum Approach.
 - Virtual Nodes in a cycle.
 - Hash both objects and caches.
 - Easy Replication.
 - Eventually Consistent.
 - What happens if nodes fail?
 - How do you add nodes?



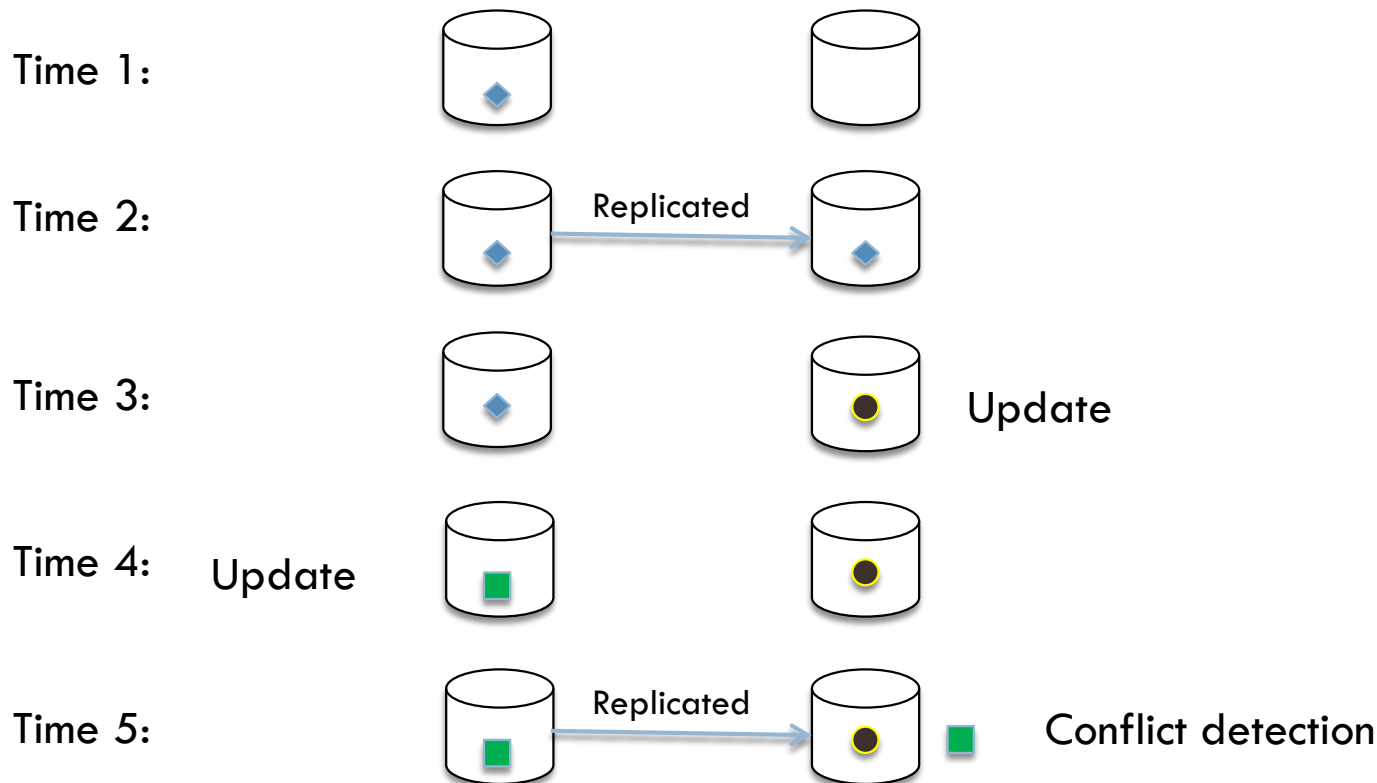
Concurrency models

- ❑ Optimistic concurrency.
- ❑ Pessimistic concurrency.
- ❑ MVCC.

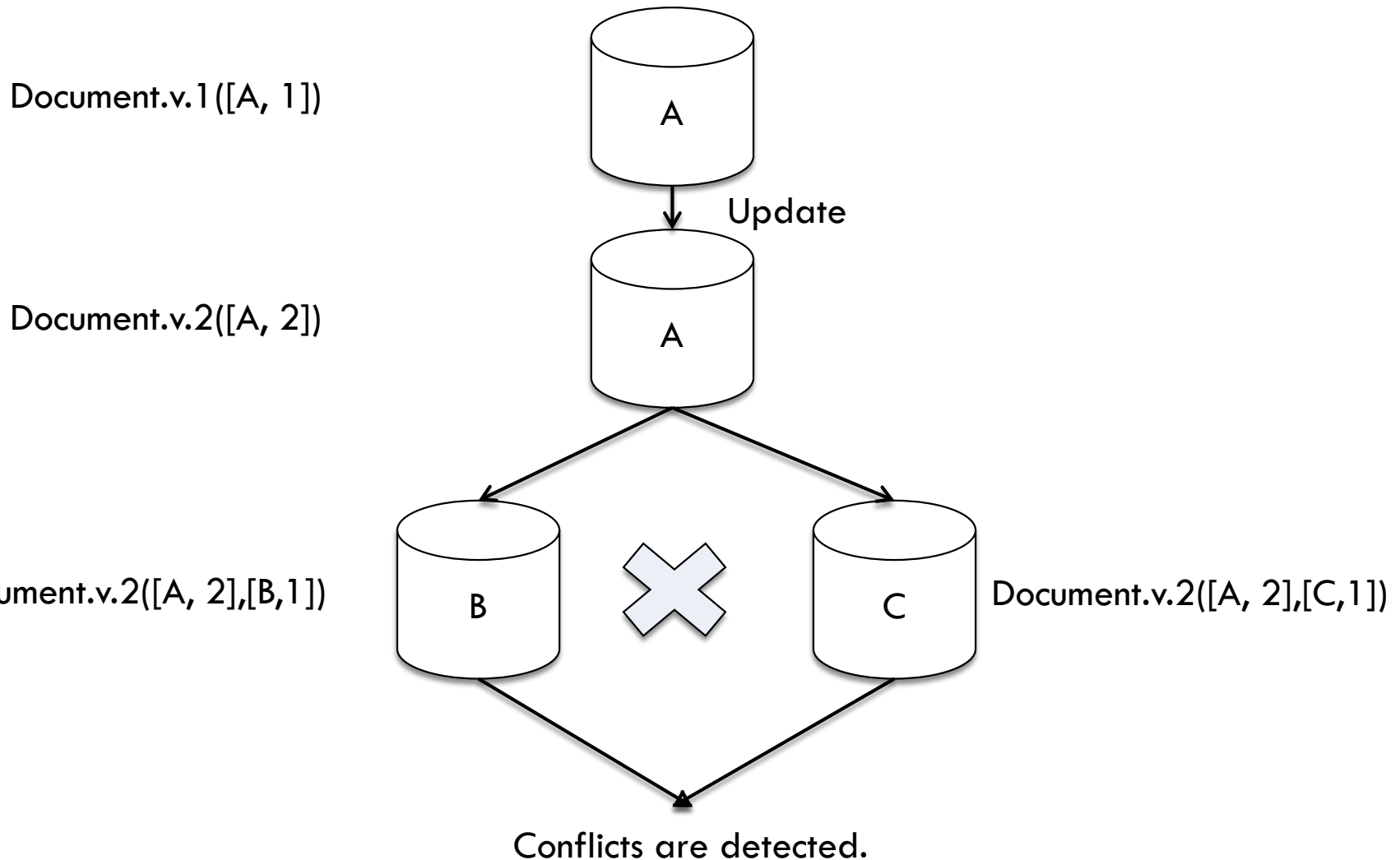


Vector Clocks

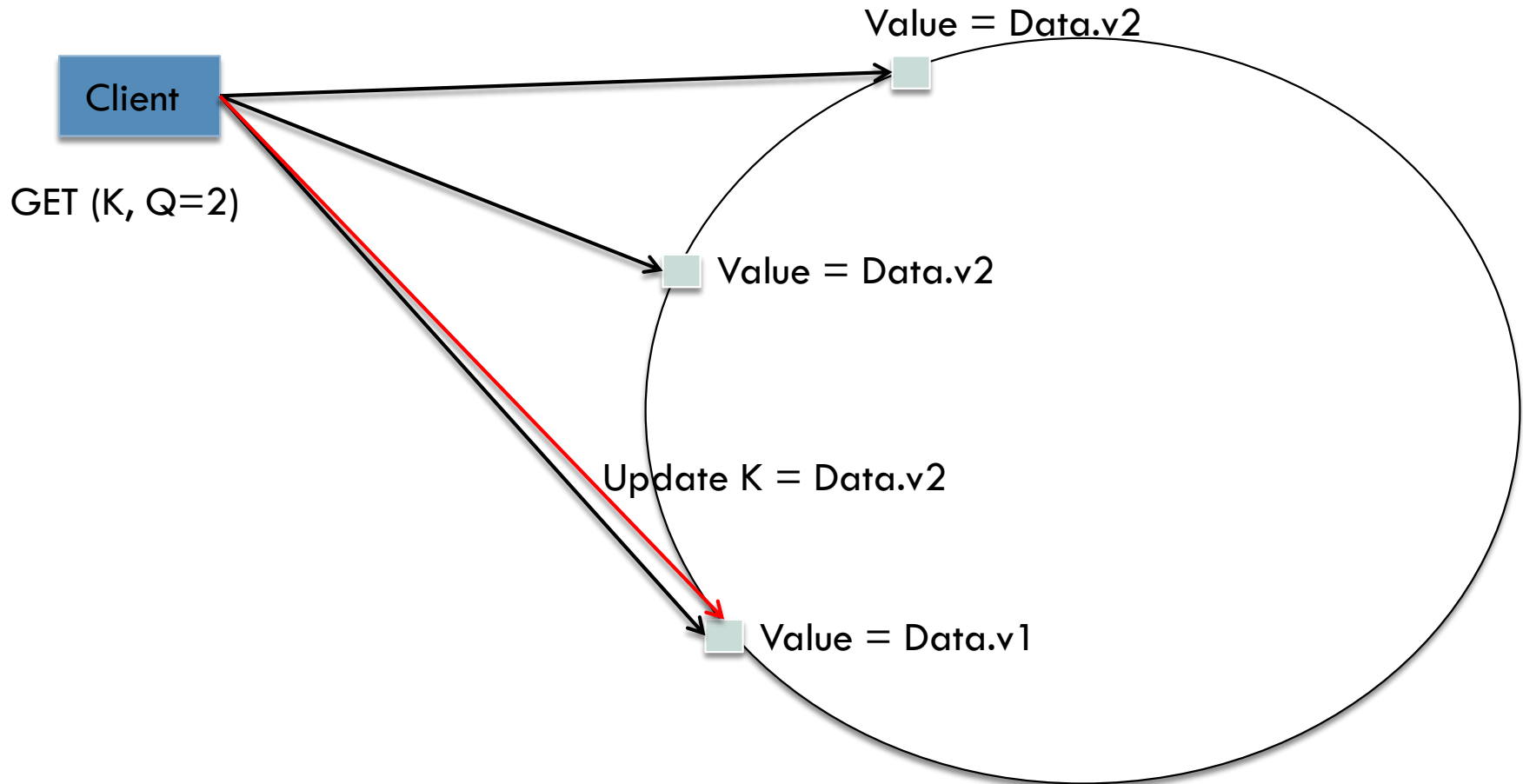
- Used for conflict **detection** of data.
- Timestamp based resolution of conflicts is not enough.



Vector Clocks

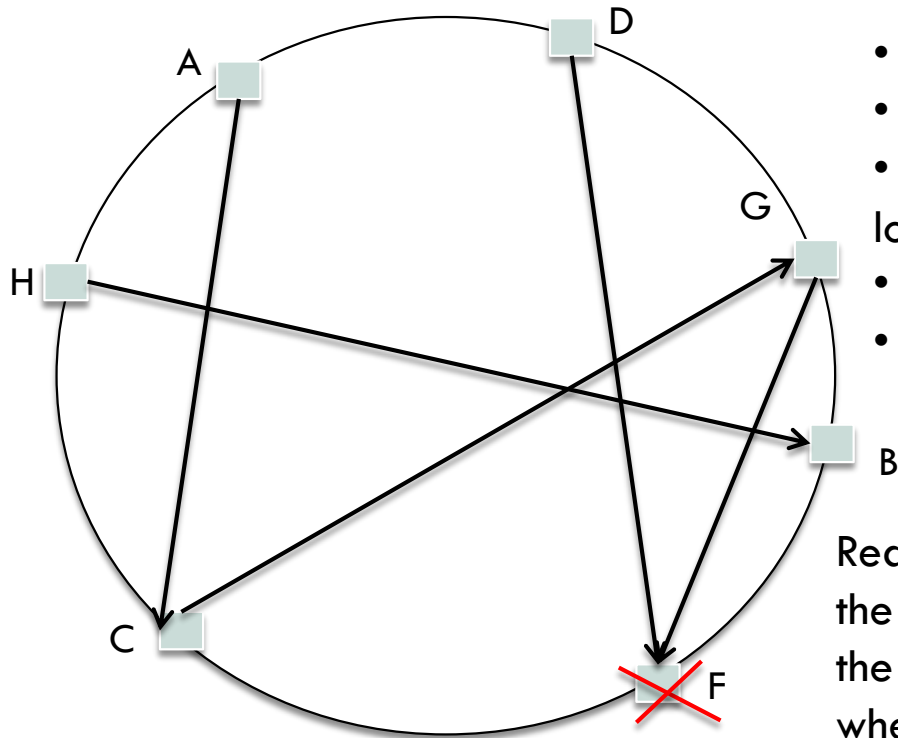


Read Repair



Gossip Protocol & Hinted Handoffs

- Most preferred communication protocol in a distributed environment is Gossip Protocol.



- All the nodes talk to each other peer wise.
- There is no global state.
- No single point of coordinator.
- If one node goes down and there is a Quorum load for that node is shared among others.
- Self managing system.
- If a new node joins, load is also distributed.

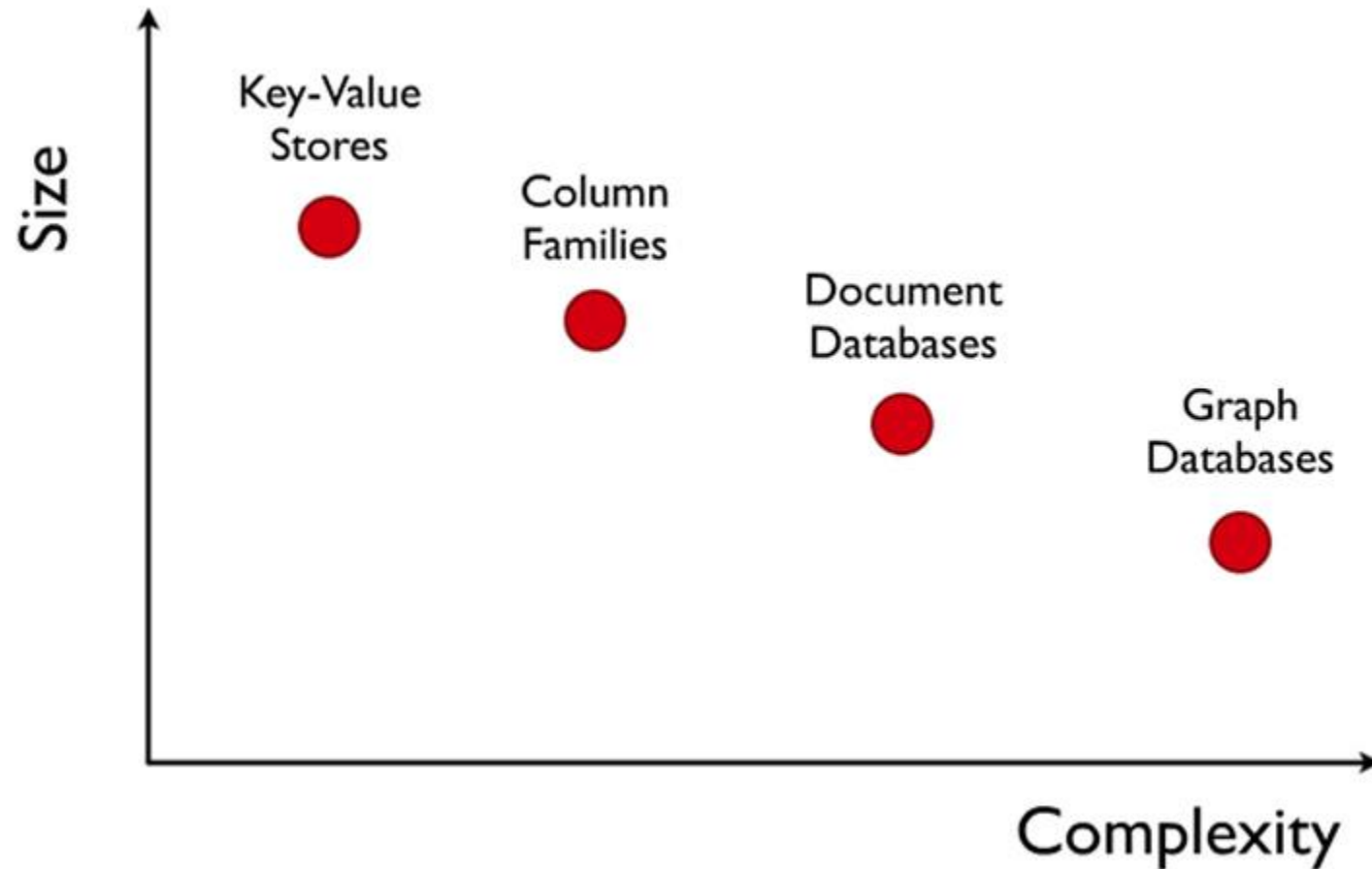
Requests coming to F will be handled by the nodes who takes the load of F, lets say C with the **hint** that it took the requests which was for F, when F becomes available, F will get this Information from C. Self healing property.

Data Models

- Key/Value Pairs.
- Tuples (rows).
- Documents.
- Columns.
- Objects.
- Graphs.

There are corresponding data stores.

Complexity



Key-Value Stores

- Memcached – Key value stores.
- Membase – Memcached with persistence and improved consistent hashing.
- AppFabric Cache – Multi region Cache.
- Redis – Data structure server.
- Riak – Based on Amazon's Dynamo.
- Project Voldemort – eventual consistent key value stores, auto scaling.

Memcached

- ❑ Very easy to setup and use.
- ❑ Consistent hashing.
- ❑ Scales very well.
- ❑ In memory caching, no persistence.
- ❑ LRU eviction policy.
- ❑ $O(1)$ to set/get/delete.
- ❑ Atomic operations set/get/delete.
- ❑ No iterators, or very difficult.

Membase

- ❑ Easy to manage via web console.
- ❑ Monitoring and management via Web console .
- ❑ Consistency and Availability.
- ❑ Dynamic/Linear Scalability, add a node, hit join to cluster and rebalance.
- ❑ Low latency, high throughput.
- ❑ Compatible with current Memcached Clients.
- ❑ Data Durability, persistent to disk asynchronously.
- ❑ Rebalancing (Peer to peer replication).
- ❑ Fail over (Master/Slave).
- ❑ vBuckets are used for consistent hashing.
- ❑ $O(1)$ to set/get/delete.

Redis

- ❑ Distributed Data structure server.
- ❑ Consistent hashing at client.
- ❑ Non-blocking I/O, single threaded.
- ❑ Values are binary safe strings: byte strings.
- ❑ String : Key/Value Pair, set/get. $O(1)$ many string operations.
- ❑ Lists: lpush, lpop, rpush, rpop. you can use it as stack or queue. $O(1)$. Publisher/Subscriber is available.
- ❑ Set: Collection of Unique elements, add, pop, union, intersection etc. set operations.
- ❑ Sorted Set: Unique elements sorted by scores. $O(\log n)$. Supports range operations.
- ❑ Hashes: Multiple Key/Value pairs
 - HMSET user 1 username foo password bar age 30
 - HGET user 1 age

Microsoft AppFabric

- ❑ Add a node to the cluster easily. Elastic scalability.
- ❑ Namespaces to organize different caches.
- ❑ LRU Eviction policy.
- ❑ Timeout/Time to live is default to 10 min.
- ❑ No persistence.
- ❑ $O(1)$ to set/get/delete.
- ❑ Optimistic and pessimistic concurrency.
- ❑ Supports tagging.

Document Stores

- ❑ Schema Free.
- ❑ Usually JSON like interchange model.
- ❑ Query Model: JavaScript or custom.
- ❑ Aggregations: Map/Reduce.
- ❑ Indexes are done via B-Trees.

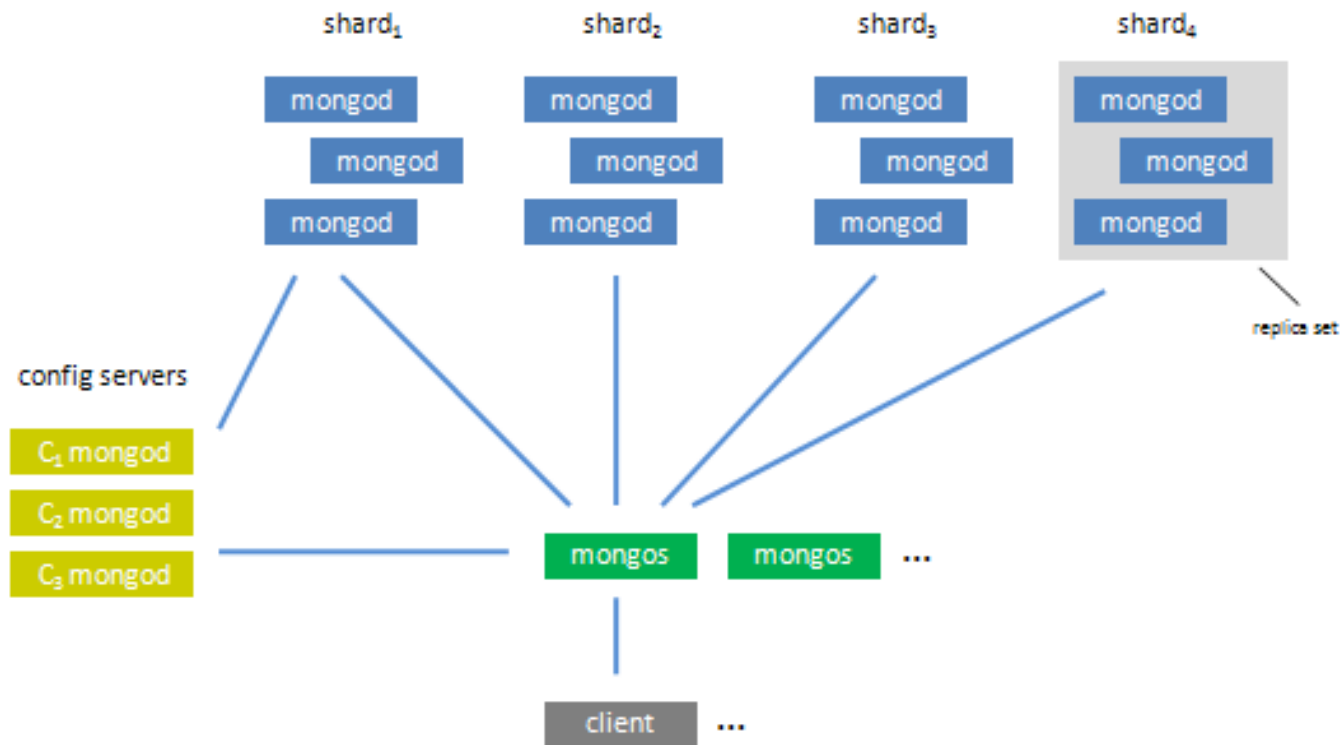
Mongodb

- ❑ Data types: bool, int, double, string, object(bson), oid, array, null, date.
- ❑ Database and collections are created automatically.
- ❑ Lots of Language Drivers.
- ❑ Capped collections are fixed size collections, buffers, very fast, FIFO, good for logs. No indexes.
- ❑ Object id are generated by client, 12 bytes packed data. 4 byte time, 3 byte machine, 2 byte pid, 3 byte counter.
- ❑ Possible to refer other documents in different collections but more efficient to embed documents.
- ❑ Replication is very easy to setup. You can read from slaves.

Mongodb

- ❑ Connection pooling is done for you. Sweet.
- ❑ Supports aggregation.
 - ❑ Map Reduce with JavaScript.
- ❑ You have indexes, B-Trees. Ids are always indexed.
- ❑ Updates are atomic. Low contention locks.
- ❑ Querying mongo done with a document:
 - ❑ Lazy, returns a cursor.
 - ❑ Reduceable to SQL, select, insert, update limit, sort etc.
 - There is more: upsert (either inserts or updates)
 - ❑ Several operators:
 - \$ne, \$and, \$or, \$lt, \$gt, \$incr, \$decr and so on.
- ❑ Repository Pattern makes development very easy.

Mongodb - Sharding



Config servers: Keeps mapping

Mongos: Routing servers

Mongod: master-slave replicas

Couchdb

- ❑ Availability and Partial Tolerance.
- ❑ Views are used to query. Map/Reduce.
- ❑ MVCC – Multiple Concurrent versions. No locks.
 - ❑ A little overhead with this approach due to garbage collection.
 - ❑ Conflict resolution.
- ❑ Very simple, REST based. Schema Free.
- ❑ Shared nothing, seamless peer based Bi-Directional replication.
- ❑ Auto Compaction. Manual with Mongoddb.
- ❑ Uses B-Trees
- ❑ Documents and indexes are kept in memory and flushed to disc periodically.
- ❑ Documents have states, in case of a failure, recovery can continue from the state documents were left.
- ❑ No built in auto-sharding, there are open source projects.
- ❑ You can't define your indexes.

Object Stores

- Objectivity.
- Db4o.

Objectivity

- ❑ No need for ORM. Closer to OOP.
- ❑ Complex data modeling.
- ❑ Schema evolution.
- ❑ Scalable Collections: List, Set, Map.
- ❑ Object relations.
 - ▣ Bi-Directional relations
- ❑ ACID properties.
- ❑ Blazingly fast, uses paging.
- ❑ Supports replication and clustering.

Column Stores

Row oriented

Id	username	email	Department
1	John	john@foo.com	Sales
2	Mary	mary@foo.com	Marketing
3	Yoda	yoda@foo.com	IT

Column oriented

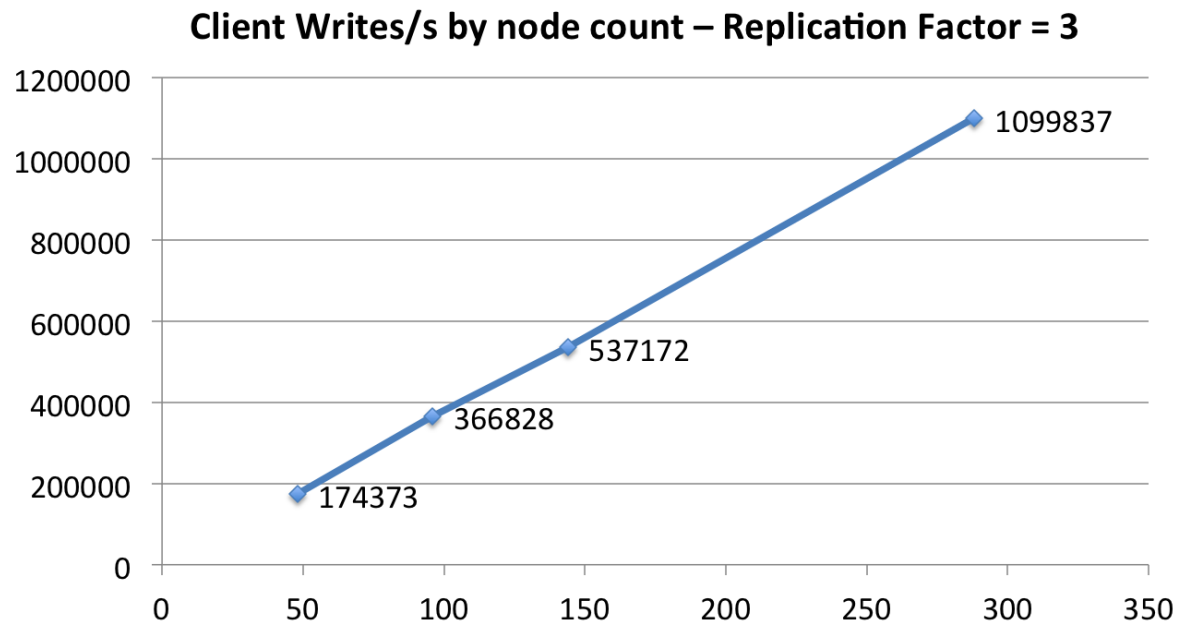
Id	Username	email	Department
1	John	john@foo.com	Sales
2	Mary	mary@foo.com	Marketing
3	Yoda	yoda@foo.com	IT

Cassandra

- ❑ Tunable consistency.
- ❑ Decentralized.
- ❑ Writes are faster than reads.
- ❑ No Single point of failure.
- ❑ Incremental scalability.
- ❑ Uses consistent hashing (logical partitioning) when clustered.
- ❑ Hinted handoffs.
- ❑ Peer to peer routing(ring).
- ❑ Thrift API.
- ❑ Multi data center support.

Cassandra at Netflix

Scale-Up Linearity



Graph Stores

- Based on Graph Theory.
- Scale vertically, no clustering.
- You can use graph algorithms easily.



Neo4J

- ❑ Nodes, Relationship.
- ❑ Traversals.
- ❑ HTTP/REST.
- ❑ ACID.
- ❑ Web Admin.
- ❑ Not too much support for languages.
- ❑ Has transactions.

Which one to use?

- Key-value stores:
 - ▣ Processing a constant stream of small reads and writes.
- Document databases:
 - ▣ Natural data modeling. Programmer friendly. Rapid development. Web friendly, CRUD.
- RDMBS:
 - ▣ OLTP. SQL. Transactions. Relations.
- OODBMS
 - ▣ Complex object models.
- Data Structure Server:
 - ▣ Quirky stuff.
- Columnar:
 - ▣ Handles size well. Massive write loads. High availability. Multiple-data centers. MapReduce
- Graph:
 - ▣ Graph algorithms and relations.

SQL vs NoSQL

In The Beginning

- ❑ Programmers processed files directly
- ❑ You would open a file, seek and read its contents, and parse out a record (same for insert / update / delete)
- ❑ Obviously, a lot of similar code was written, and eventually refactored into libraries
- ❑ These libraries became databases, each with their own strengths and weaknesses, and their own APIs
- ❑ IBM came up with the idea for SEQUEL (Structured English QUery Language) around 1970, and was implemented in several database libraries

RDBMS

- Relational DataBase Management System
- Main Focus is ACID
 - ▣ Atomicity – Each transaction is atomic. If one part of it fails, the entire transaction fails (and is rolled back)
 - ▣ Consistency – Every transaction is subject to a consistent set of rules (constraints, triggers, cascades)
 - ▣ Isolation – No transaction should interfere with another transaction
 - ▣ Durability – Once a transaction is committed, it remains committed

RDBMS

- ACID is important
 - ▣ But only when it's important
 - ▣ i.e. banking, finance, safety systems, etc.
 - ▣ The kinds of systems that people were building with computers 30 years ago (and today)
- ACID adds overhead
 - ▣ Features like atomicity, isolation basically force database servers to use sequential evaluation
 - ▣ Add a concept like multiple-servers?

RDBMS

- Other Features
 - ▣ Typed columns
 - ▣ Normalized data
 - ▣ Supports broad, unrestrained queries
 - ▣ Has its own query optimizer
 - ▣ Consistent way of accessing data (SQL)

SQL

- A common (for the most part) query language for RDBMS
- Enables unrestrained queries against normalized data
- Geared towards joins, filters, and aggregations
- Relies heavily on the database server's query optimization
 - ▣ Sometimes requires the involvement of a DBA

The CAP Theorem

- ❑ Impossible for any shared data-system to guarantee simultaneously all of the following three properties:
 - ▣ Consistency – once data is written, all future read requests will contain that data
 - ▣ Availability – the database is always available and responsive
 - ▣ Partition Tolerance – if part of the database is unavailable, other parts are unaffected

Good Quote!

Google, Amazon, Facebook, and DARPA all recognized that when you scale systems large enough, you can never put enough iron in one place to get the job done (and you wouldn't want to, to prevent a single point of failure). Once you accept that you have a distributed system, you need to give up consistency or availability, which the fundamental transactionality of traditional RDBMSs cannot abide.

- Cedric Beust

Enter NoSQL

- ❑ RDBMS doesn't quite fit for *some* requirements
- ❑ Google and Amazon decided to make their own stuff (aka BigTable and S3 Storage) to meet their own unique needs
 - ❑ Distributed
 - ❑ Scalability
 - ❑ Control over performance characteristics
 - ❑ High availability
 - ❑ Low Latency
 - ❑ Cheap

What is NoSQL?

- Basically a large serialized object store*
 - ▣ (mostly) retrieve objects by defined ID
- In general, doesn't support complicated queries*
- Doesn't have a structured (or any!) schema*
 - ▣ Recommends denormalization
- Designed to be distributed (cloud-scale) out of the box
- Because of this, drops the ACID requirements
 - ▣ Any database can answer any query
 - ▣ Any write query can operate against any database and will “eventually” propagate to other distributed servers

* Dependent on vendor

The opposite of ACID is...

□ BASE

- ▣ Basically Available – guaranteed availability
- ▣ Soft-state – the state of the system may change, even without a query (because of node updates)
- ▣ Eventually Consistent – the system will become consistent over time

□ Contrived acronym, but so is ACID :P

NoSQL – Eventual Consistency

- Because of the distributed model, any server can answer any query
- Servers communicate amongst themselves at their own pace (behind the scenes)
- The server that answers your query might not have the latest data
- Who really cares if you see Kim Kardashian's latest tweet instantaneously?

Different Types of NoSQL

- Column Store
 - ▣ Column data is saved together, as opposed to row data
 - ▣ Super useful for data analytics
 - ▣ Hadoop, Cassandra, Hypertable
- Key-Value Store
 - ▣ A key that refers to a payload
 - ▣ MemcacheDB, Azure Table Storage, Redis
- Document / XML / Object Store
 - ▣ Key (and possibly other indexes) point at a serialized object
 - ▣ DB can operate against values in document
 - ▣ MongoDB, CouchDB, RavenDB
- Graph Store
 - ▣ Nodes are stored independently, and the relationship between nodes (edges) are stored with data

Also Important – MapReduce

- One of the most common large queries (and hard to optimize) is Map Reduce
 - ▣ Map: Select a subset of data (into a key, value pair)
 - ▣ Reduce: Perform some sort of operation on that data
 - ▣ Used primarily for data analysis (reporting)
 - ▣ Example: Average revenue on all invoices in 2010
- In RDBMS, this is usually pretty expensive
 - ▣ Involves a full table scan with a select
 - ▣ If it's not a built-in aggregate function, involves custom code on database server (sproc/udf)
- In (some) NoSQL, it's automatically distributed
 - ▣ MapReduce functions are automatically distributed across all nodes to process, and the result is automatically gathered and returned as part of the vendor's framework
- In Google's environment, can run a MapReduce operation across all their nodes in about 30 minutes.

NoSQL, Mo' Problems

- ❑ Inconsistent APIs between NoSQL providers
- ❑ Denormalized data requires you to maintain your own data relationships (cascades) in code
- ❑ Not a lot of real operational power for DevOps / IT
- ❑ Lack of complicated queries requires joins / aggregations / filters to be done in code (except for MapReduce)
 - ▣ Arguably not a bad thing?

That being said...

NoSQL-type databases power:

- Google reader
- Google maps
- Blogger.com
- Youtube
- Gmail
- Amazon
- Sourceforge
- Github
- CollegeHumor
- Justin.tv
- Grooveshark
- BMW
- Cisco
- Honda
- Mozilla
- Adobe
- Ebay
- Facebook
- Hulu
- Last.Fm
- LinkedIn
- New York Times
- Twitter
- Yahoo!
- Disney
- Craigslist
- Foursquare
- Forbes
- Bit.ly
- Intuit
- Boeing
- US Army
- Seagate
- Hertz
- IBM
- Intel
- Oxford University Press
- United Airways
- University of Toronto
- XQ ☺

In Conclusion!

- RDBMS is a great tool for solving ACID problems
 - ▣ When data validity is super important
 - ▣ When you need to support dynamic queries
- NoSQL is a great tool for solving data availability problems
 - ▣ When it's more important to have fast data than right data
 - ▣ When you need to scale based on changing requirements
- Pick the right tool for the job

Column Based Database

Columnar Databases Overview

- A column-oriented DBMS is a database management system (DBMS) that stores its content by **column** rather than by row. This has advantages for data warehouses and library catalogues where aggregates are computed over large numbers of similar data items.
- A column-oriented database serializes all of the values of a column together, then the values of the next column, and so on. In a column-oriented database, only the columns in the query need to be retrieved.
- Advantage of column oriented databases over row oriented databases is in the efficiency of hard-disk access.

Disadvantage of a Row Based Database

- ❑ In a RDBMS, data values are collected and managed as individual rows and events containing related rows.
- ❑ A row-oriented database must read the entire record or “row” in order to access the needed attributes or column data.
- ❑ Queries most often end up reading significantly more data than is needed to satisfy the request and it creates very large I/O burdens.
- ❑ Architects and DBAs often tune the environment for the different queries by building additional indexes, pre-aggregating data, and creating special materialized views and cubes. Resulted in more processing time and consume additional persistent data storage.
- ❑ Most of the tuning are query-specific, these tunings only address the performance of the queries that are known, and do not even touch upon general performance of ad hoc queries.

[illegible]

One difference that Matters Most

The major significant difference between columnar and row-based stores is that all the columns of a table are not stored successively in storage – in the data pages. This eliminates much of the metadata that is stored on a data page, which helps the data management component of the DBMS navigate the many columns in a row-based database as quickly as it can. In a relational, row-based page, there is a map of offsets near the end of the page to where the records start on the page. This map is updated as records come and go on the page. The offset number is also an important part of how an index entry would find the rest of the record in the data page in a row-based system. The need for indexes is greatly minimized in column-based systems, to the point of not being offered in many columnar Databases.

Columnar Database

- A column-oriented database has its data organized and stored by columns.
- System can evaluate which columns are being accessed and retrieve only the values requested from the specific columns.
- The data values themselves within each column form the index, reducing I/O, enabling rapid access.

Block 1	7369	SMITH	CLERK	7902	17/12/2000
Block 2	7499	ALLEN	SALESMAN	7698	20/02/2001
Block 3	7521	WARD	SALESMAN	7698	22/02/2001

Row Database stores row values together

EmpNo	EName	Job	Mgr	HireDate
7369	SMITH	CLERK	7902	17/12/1980
7499	ALLEN	SALESMAN	7698	20/02/1981
7521	WARD	SALESMAN	7698	22/02/1981
7566	JONES	MANAGER	7839	2/04/1981
7654	MARTIN	SALESMAN	7698	28/09/1981
7698	BLAKE	MANAGER	7839	1/05/1981
7782	CLARK	MANAGER	7839	9/06/1981

Block 1	7369	7499	7521	7566	7654
Block 2	SMITH	ALLEN	WARD	JONES	MARTIN
Block 3	CLERK	SALESMAN	SALESMAN	MANAGER	SALESMAN

Column Database stores column values together

Row Store Physical layout

Logical Schema

Column Store physical layout

Food for Thought !!!!

Think about it from an efficiency standpoint. When I want just a few songs from an album, it's cheaper to purchase only those songs from iTunes that I want. When I want most of the songs, I will save a couple bucks by purchasing the whole album. Over time, I may find that I like one of those songs.

However, when it comes to a query, the query either wants a column or it doesn't. It will not come to like a column later that it was forced to select. This is foundational to the value proposition for columnar databases.....

Benefits of a Columnar Database

- Better analytic performance: row oriented approach allows better performance in running a large number of simultaneous queries.
- Rapid joins and aggregation: data access streaming along column-oriented data allows for incrementally computing the results of aggregate functions, which is critical for data warehouse applications.
- Suitability for compression: Eliminates storage of multiple indexes, views and aggregations, and facilitates vast improvements in compression.
- Rapid data loading: In a columnar arrangement the system effectively allows one to segregate storage by column. This means that each column is built in one pass, and stored separately, allowing the database system to load columns in parallel using multiple threads. Further, related performance characteristics of join processing built atop a column store is often sufficiently fast that the load-time joining required to create fact tables is unnecessary, shortening the latency from receipt of new data to availability for query processing. Finally, since columns are stored separately, entire table columns can be added and dropped without downing the system, and without the need to re-tuning the system following the change

Challenges

- ❑ No one-size-fits-all system.
- ❑ Load time: Converting the data source into columnar format can be unbearably slow where tens or hundreds of gigabytes of data are involved.
- ❑ Incremental loads: Incremental loads can be performance problematic.
- ❑ Data compression: Some columnar systems greatly compress the source data. However, uncompressing the data to read it can slow performance.
- ❑ Structural limitations: Columnar databases use different techniques to simulate a relational structure. Some require the same primary key on all tables, meaning the database hierarchy is limited to two levels. The limits imposed by a particular system may not seem to matter, but remember that your needs may change tomorrow. Constraints that seem acceptable now could prevent you from expanding the system in the future.
- ❑ Scalability: Columnar databases major advantage is to get good performance on large databases. However, is there is reasonable to use columnar databases in case you are dealing with common size database?

Best Practices in using Columnar DB

□ **Use to Save Money on Storage**

- All column data stores keep the data in the same row order so that when the records are pieced together, the correct concatenation of columns is done to make up the row.
- matches values to rows according to the position of the value (i.e., 3rd value in each column belongs to the 3rd row, etc.). This way "SUVRADDEEP" (from the first name column file2) is matched with "RUDRA" (from the last name column file) correctly – instead of matching "DAS" with "SUVRADDEEP", for example.
- Dictionary Method - dictionary structure is used to store the actual values along with tokens .
- The dictionary arrangement allows DB, to trim insignificant trailing nulls from character fields, furthering the space savings. Effectively, characters over 8 bytes are treated as variable length characters.

For example, 1=State Grid Corporation of China, 2=Nippon Telegraph and Telephone and 3=Federal Home Loan Mortgage Corporation could be in the dictionary and when those are the column values, the 1, 2 and 3 are used in lieu of the actual values. If there are 1,000,000 customers with only 50 possible values, the entire column could be stored with 8 megabytes (8 bytes per value). The separate dictionary structure, containing each unique value and its associated token, would have more page-level metadata. Since each value can have a different length, a map to where the values start on the page would be stored, managed and utilized in page navigation.

- **DICTIONARY: 1, State Grid Corporation of China, 2, Nippon Telegraph and Telephone, 3, Federal Home Loan Mortgage Corporation**
- **DATA PAGE: 1,3,2,3,1,3,1, ...**

Speed for Input/output Bound Queries

- **Optimized the I/O operation**
- In row-based databases, complete file scans mean I/O of data that is non-essential to the query. This non-essential data could comprise a very large percentage of the I/O.
- Much more of the data in the I/O is essential to a columnar query. An I/O in a columnar database will only retrieve one column – a column interesting to the particular query from either a selection or projection (WHERE clause) capacity. The projection function starts first and gathers a list of record numbers to be returned, which is used with the selection queries (if different from projection) to materialize the result set.
- Columnar databases can perform full column scans much quicker than a row-based system would turn to a full table scan. Query time spent in the optimizer is reduced significantly.
- Columnar databases are one of many new approaches taking workloads off the star schema data warehouse, which is where many of the I/O bound queries are today. Heterogeneity in post-operational systems is going to be the norm for some time, and columnar databases are a major reason because they can outperform many of the queries executed in the data warehouse.

Beyond Cubes

- Multidimensional databases (MDBs), or cubes, are separate physical structures that support very fast access to selective data.

- When a query asks for most columns of the MDB, the MDB will perform quite well. The physical storage of these MDBs is a demoralized dimensional model, which eliminates joins. However, MDBs get large and grow faster than expected as columns are added. They can also grow in numbers across the organization, becoming an unintentional impediment to information access.
 - The processing step (data load) can be quite lengthy, especially on large data volumes in a MDB.
 - Difficulty updating and querying models with more than ten dimensions.
 - Traditionally have difficulty querying models with dimensions with very high cardinality

- It is difficult to develop the necessary discipline to use MDBs with its best-fit workloads. MDB abuse is a major cause of the complete overhaul of the information management environment. Many are looking for scalable alternatives and the analytic workloads used with MDBs tend to have a lot in common with the more manageable columnar databases.

Not a cup of tea for Hadoop

- Hadoop is a parallel programming framework for large-scale data.
- The ideal workload for Hadoop is data that is massive not only from the standpoint of collecting history over time, but also from the standpoint of high volume in a single day.
- Hadoop systems are flat file based with no relational database for performance, nearly all queries run a file scan for every task, even if the answer is found in the first block of disk data.
- Hadoop systems are best suited for unstructured data, for that is the data that amasses large very quickly, needing only batch processing and a basic set of query capabilities.
- It is **not** for data warehousing nor the analytic, warehousing-like workloads.
- Therefore ,summary data will be sent from Hadoop (using Sqoop ,Flume, Hive ..etc) to the **columnar database**. Analysts and business consumers access the columnar database 7 X 24 for ad-hoc reporting and analysis, whereas Hadoop access is scheduled and more restricted.

Columnar databases allow you to implement a data model free of the tuning and massaging that must occur to designs in row-based databases, such as making the tables unnaturally small to simulate columnar efficiencies.

*Sqoop - Integration of databases and data warehouses with Hadoop

*Flume - Configurable streaming data collection

*Hive - SQL-like queries and tables on large datasets

Conclusion

- Columnar database benefits are enhanced with larger amounts of data, large scans and I/O bound queries. While providing performance benefits, they also have unique abilities to compress their data, like cubes, data warehouses and Hadoop, they are an important component of a modern, heterogeneous environment. By following these guidelines and moving the best workloads to columnar databases ,an organization is best enabled to pursue the full utilization of one of its most important assets - information.

□ Introduction to

MongoDB

□ The Great Divide



MongoDB - Sweet Spot: **Easy**, **Flexible** and **Scalable**

□ What is MongoDB ?

- Scalable High-Performance Open-source, Document-orientated database.
- Built for Speed
- Rich Document based queries for **Easy readability**.
- Full Index Support for **High Performance**.
- Replication and Failover for **High Availability**.
- Auto Sharding for **Easy Scalability**.
- Map / Reduce for **Aggregation**.

□ Why use MongoDB?

- SQL was invented in the 70's to store **data**.
- MongoDB stores **documents (or) objects**.
- Now-a-days, everyone works with **objects** (Python/Ruby/Java/etc.)
- And we need Databases to persist our **objects**.
Then why not store **objects** directly ?
- Embedded documents and arrays reduce need for joins. **No Joins** and No-multi document **transactions**.

□ What is MongoDB great for?

- RDBMS replacement for **Web Applications**.
- **Semi-structured** Content Management.
- **Real-time** Analytics & High-Speed Logging.
- Caching and **High Scalability**

Web 2.0, Media, SAAS, Gaming

HealthCare, Finance, Telecom, Government

□ Not great for?

- Highly **Transactional** Applications.
- Problems requiring **SQL**.

Some Companies using MongoDB in Production



Let's Dive in !

When I say

Think

Database

Database

- Made up of Multiple **Collections**.
- Created **on-the-fly** when referenced for the first time.

When I say

Think

Collection

Table

- Schema-less, and contains **Documents**.
- **Indexable** by one/more keys.
- Created **on-the-fly** when referenced for the first time.
- **Capped Collections**: Fixed size, older records get dropped after reaching the limit.

When I say

Think

Document

Record/Row

- Stored in a **Collection**.
- Can have **_id** key – works like Primary keys in MySQL.
- Supported Relationships – **Embedded (or) References**.
- Document storage in **BSON** (Binary form of JSON).

□ Understanding the Document Model.

```
var p = {  
  '_id': '3432',  
  'author': DBRef('User', 2),  
  'title': 'Introduction to MongoDB',  
  'body': 'MongoDB is an open sources.. ',  
  'timestamp': Date('01-04-12'),  
  'tags': ['MongoDB', 'NoSQL'],  
  'comments': [{ 'author': DBRef('User', 4),  
                  'date': Date('02-04-12'),  
                  'text': 'Did you see.. ',  
                  'upvotes': 7, ... }  
]  
}  
> db.posts.save(p);
```

□ Secondary Indexes

Create Index on any field in the document

// 1 means ascending, -1 means descending

```
> db.posts.ensureIndex({'author': 1});
```

//Index Nested Documents

```
> db.posts.ensureIndex('comments.author': 1);
```

// Index on tags

```
> db.posts.ensureIndex({'tags': 1});
```

// Geo-spatial Index

```
> db.posts.ensureIndex({'author.location': '2d'});
```

□ What about Queries? So Simple

// find posts which has 'MongoDB' tag.

```
> db.posts.find({tags: 'MongoDB'});
```

// find posts by author's comments.

```
> db.posts.find({'comments.author':  
DBRef('User',2)}).count();
```

// find posts written after 31st March.

```
> db.posts.find({'timestamp': {'gte': Date('31-03-12')}});
```

// find posts written by authors around [22, 42]

```
> db.posts.find({'author.location': {'near':[22, 42]}});
```

`$gt, $lt, $gte, $lte, $ne, $all, $in, $nin, count, limit, skip, group, etc...`

□ What about Updates? **Atomic Operations** makes it simple

```
db.posts.update({_id: '3432'},  
{ 'title': 'Introduction to MongoDB (updated)',  
  'text': 'Updated text',  
  $addToSet: { 'tags': 'webinar' } });
```

\$set, \$unset

\$push, \$pull, \$pop, \$addToSet

\$inc, \$decr, many more...

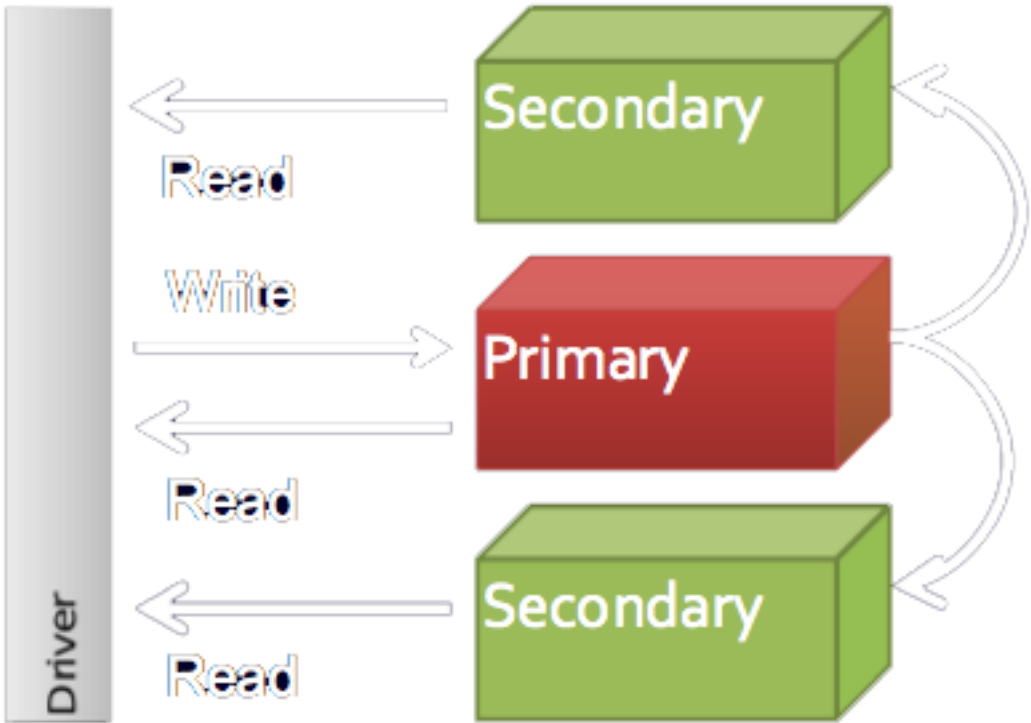
Where are my joins and transactions? !!!

□ Some Cool features

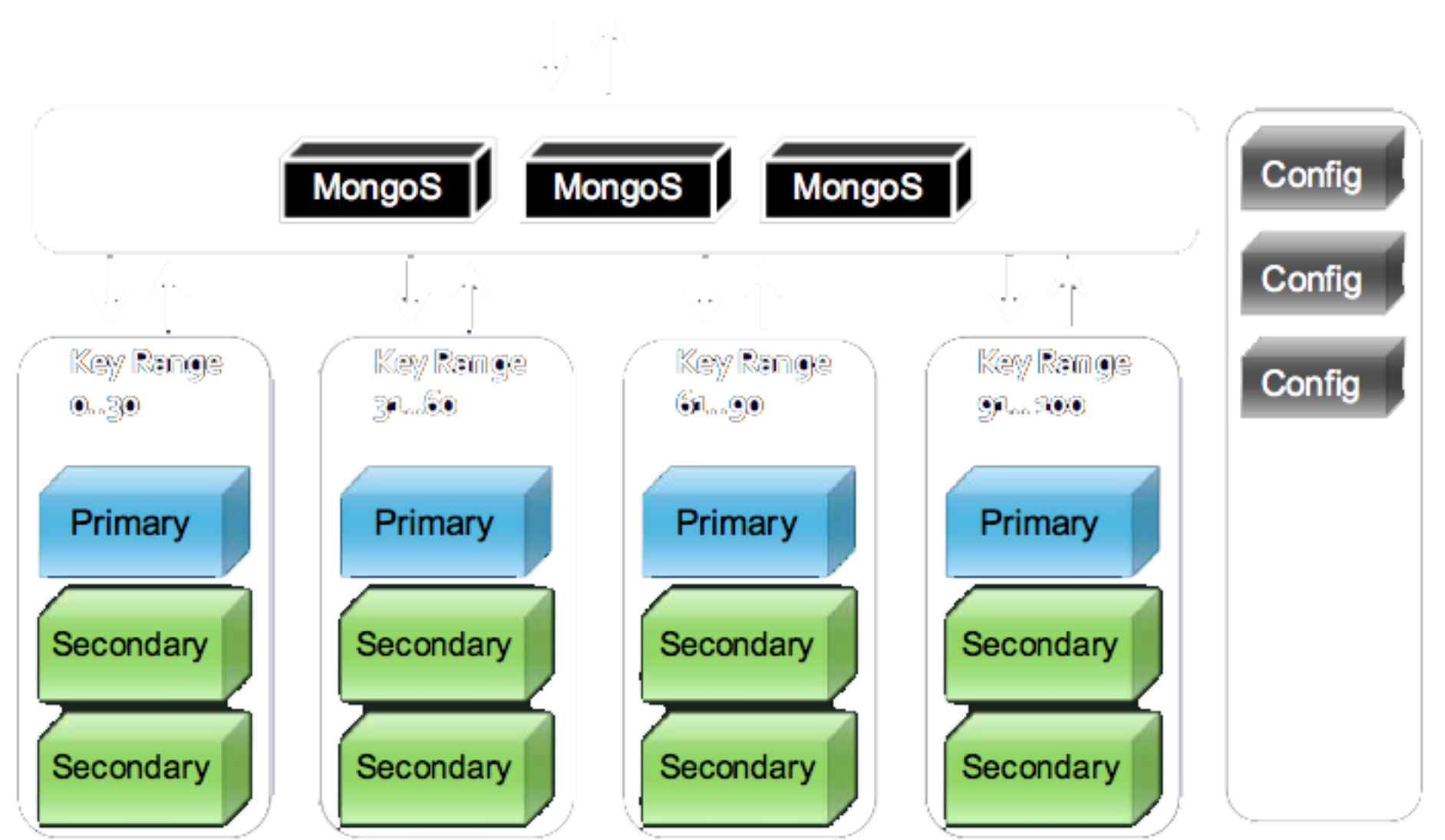
- Geo-spatial Indexes for Geo-spatial queries.
\$near, \$within_distance, Bound queries (circle, box)
- GridFS
Stores Large Binary Files.
- Map/Reduce
GROUP BY in SQL, map/reduce in MongoDB.

Deployment & Scaling

□ Replica Sets



Sharding

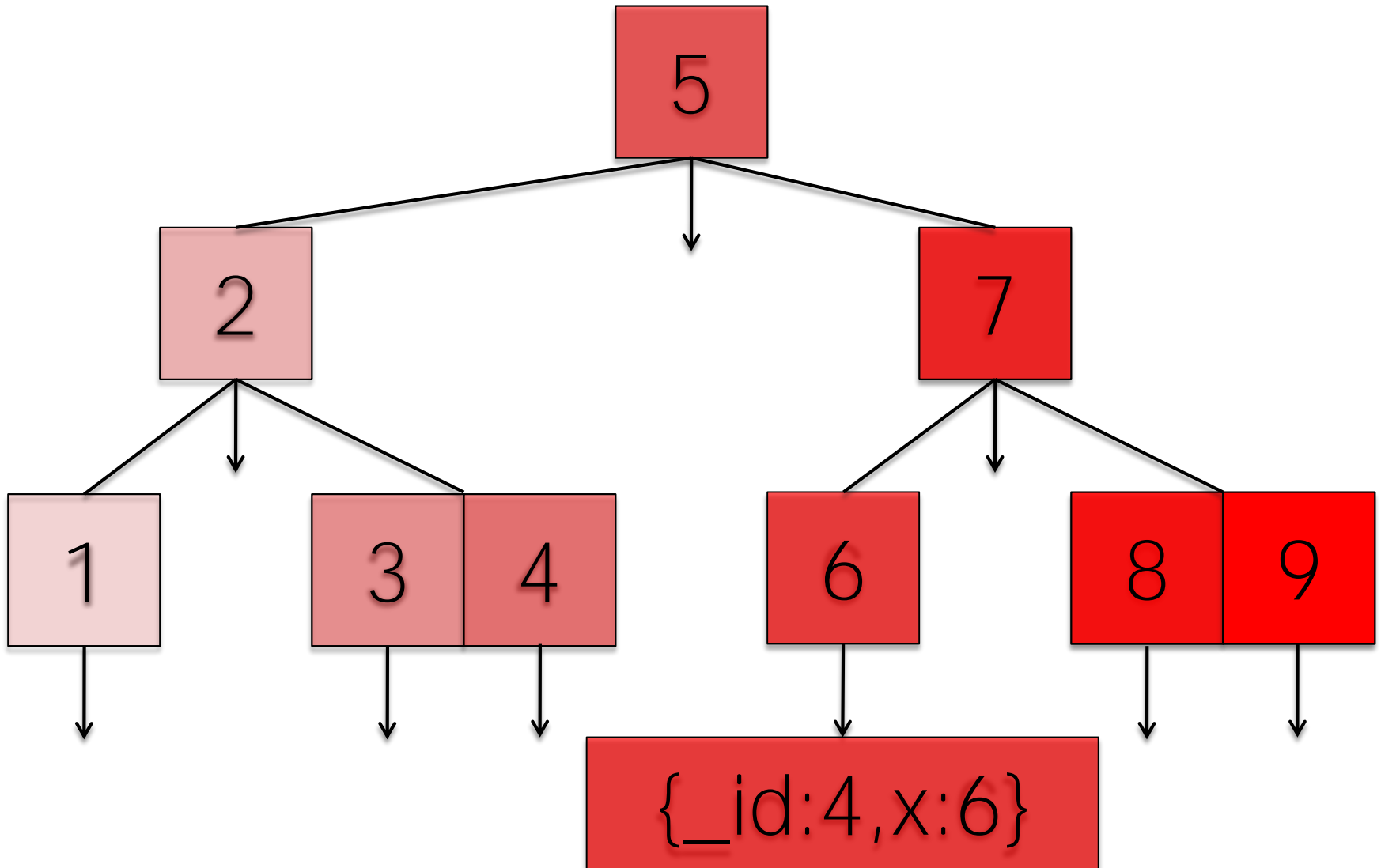


MongoDB

Indexing and Query Optimizer

Details

Btree (just a conceptual diagram)



Basic Index Bounds

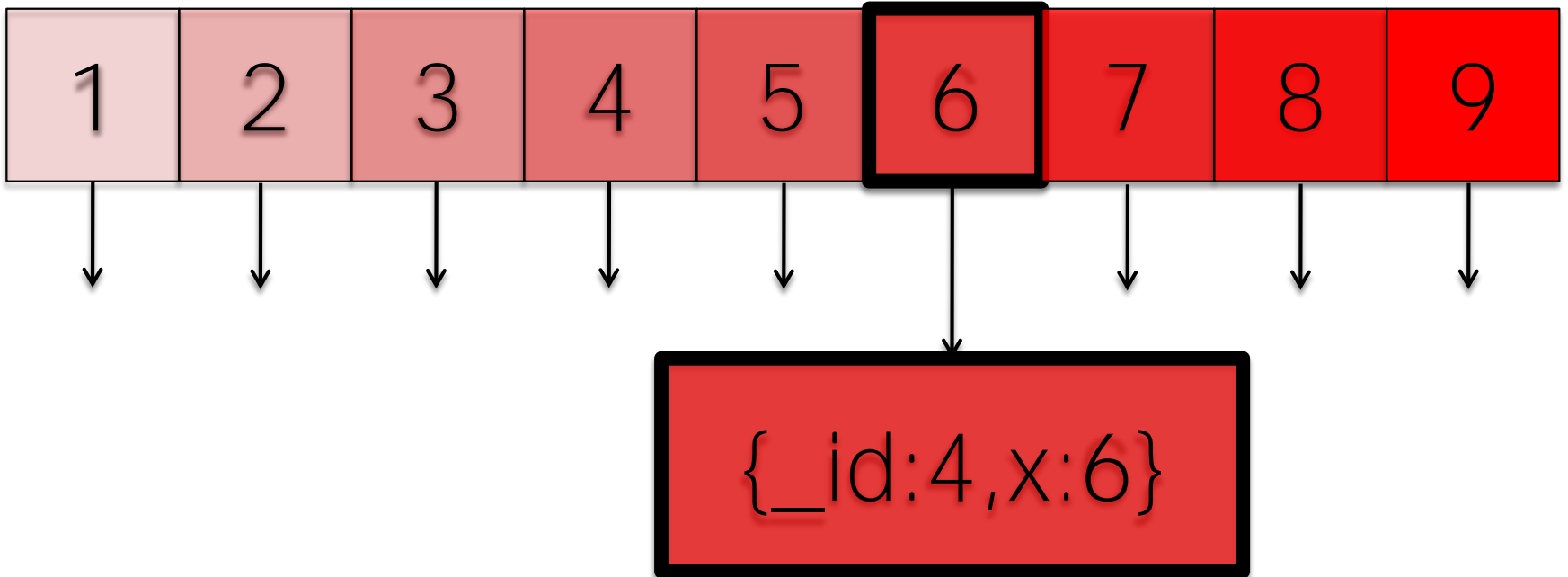
Find One Document

- `db.c.find({x:6}).limit(1)`
- Index `{x:1}`

Find One Document

6

?



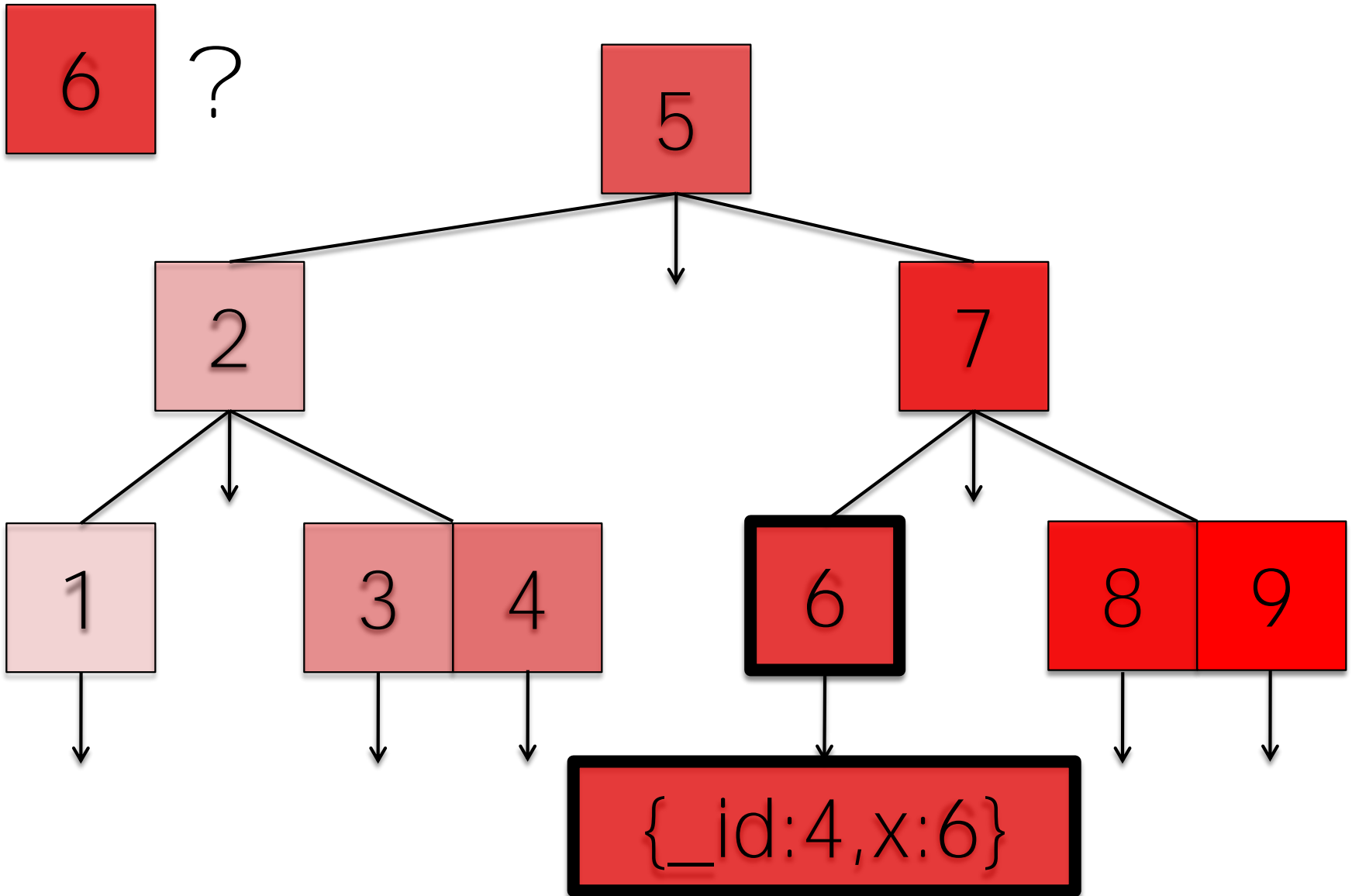
Find One Document

"nscanned" : 1,

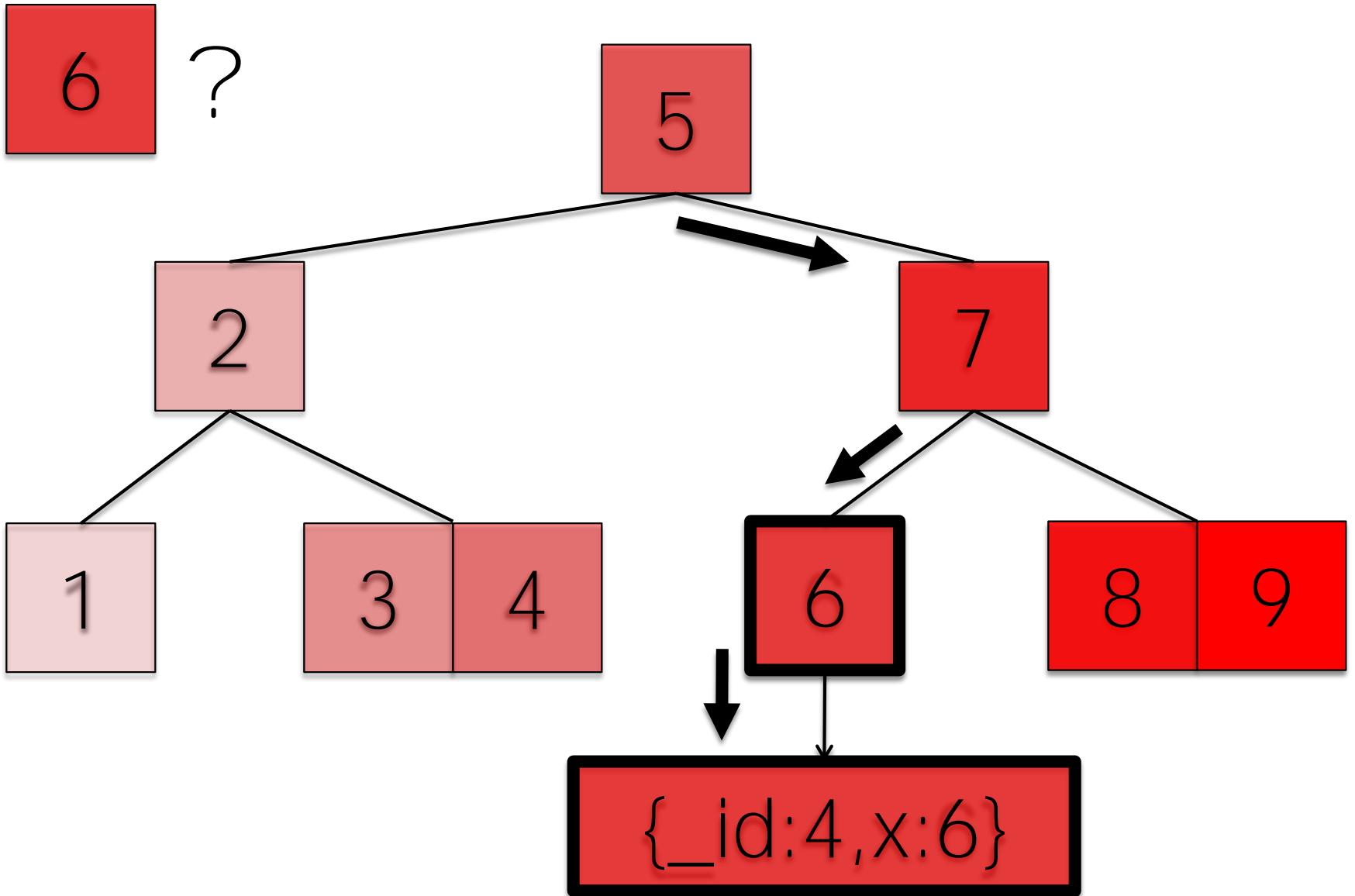
"nscannedObjects" : 1,

"n" : 1,

Find One Document



Find One Document



Find One Document

6

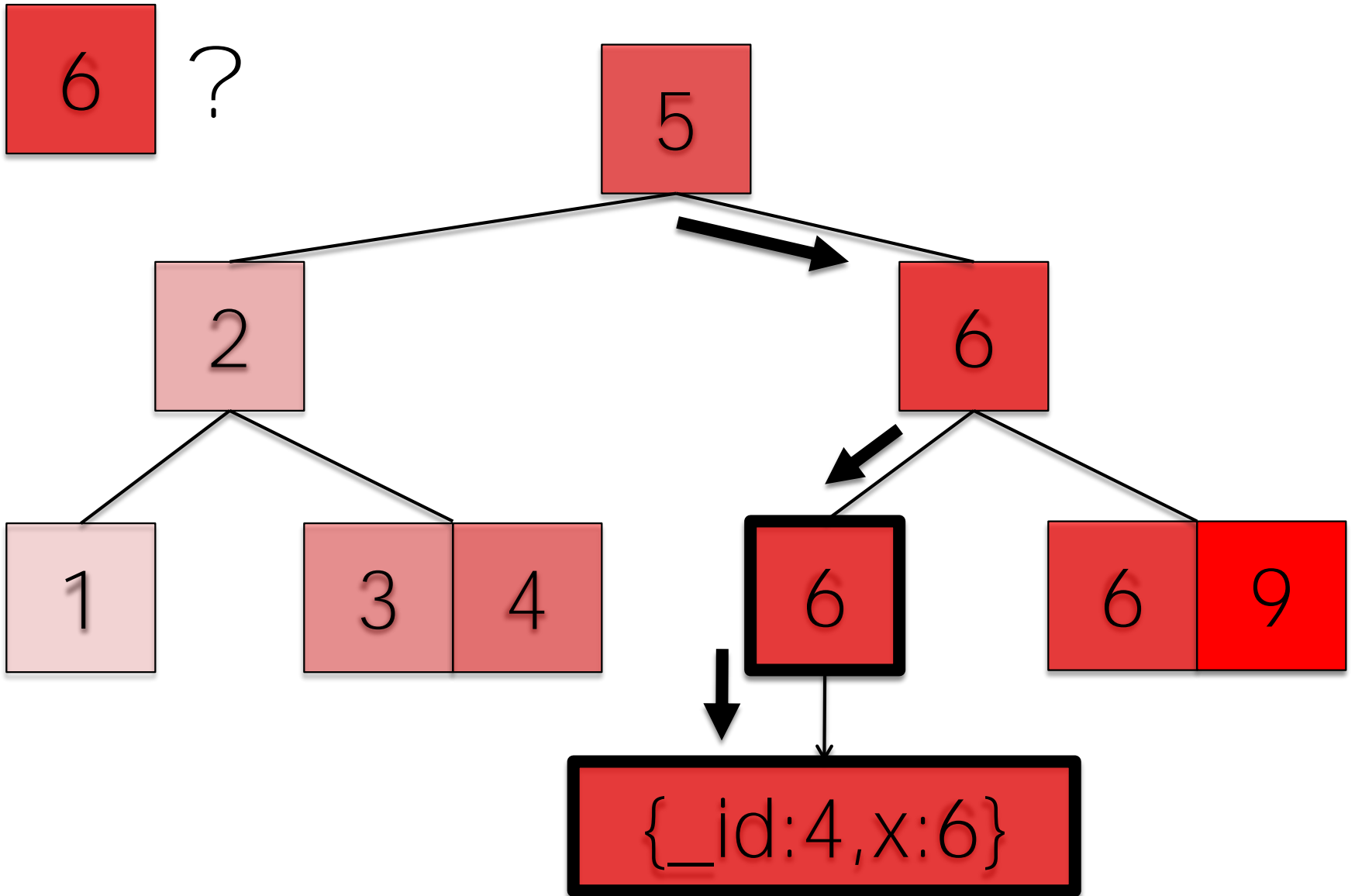
?

Now we have
duplicate x values



{_id:4,x:6}

Find One Document

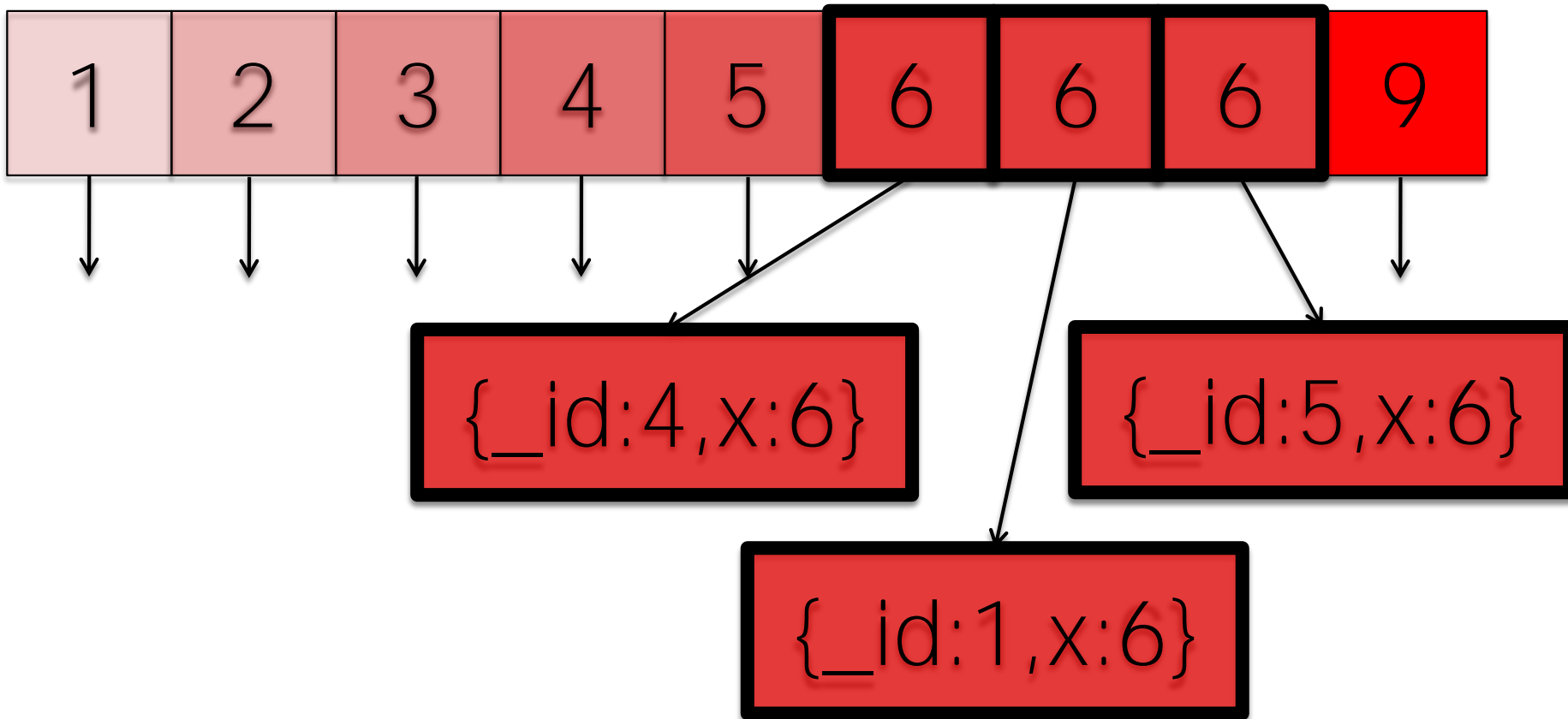


Equality Match

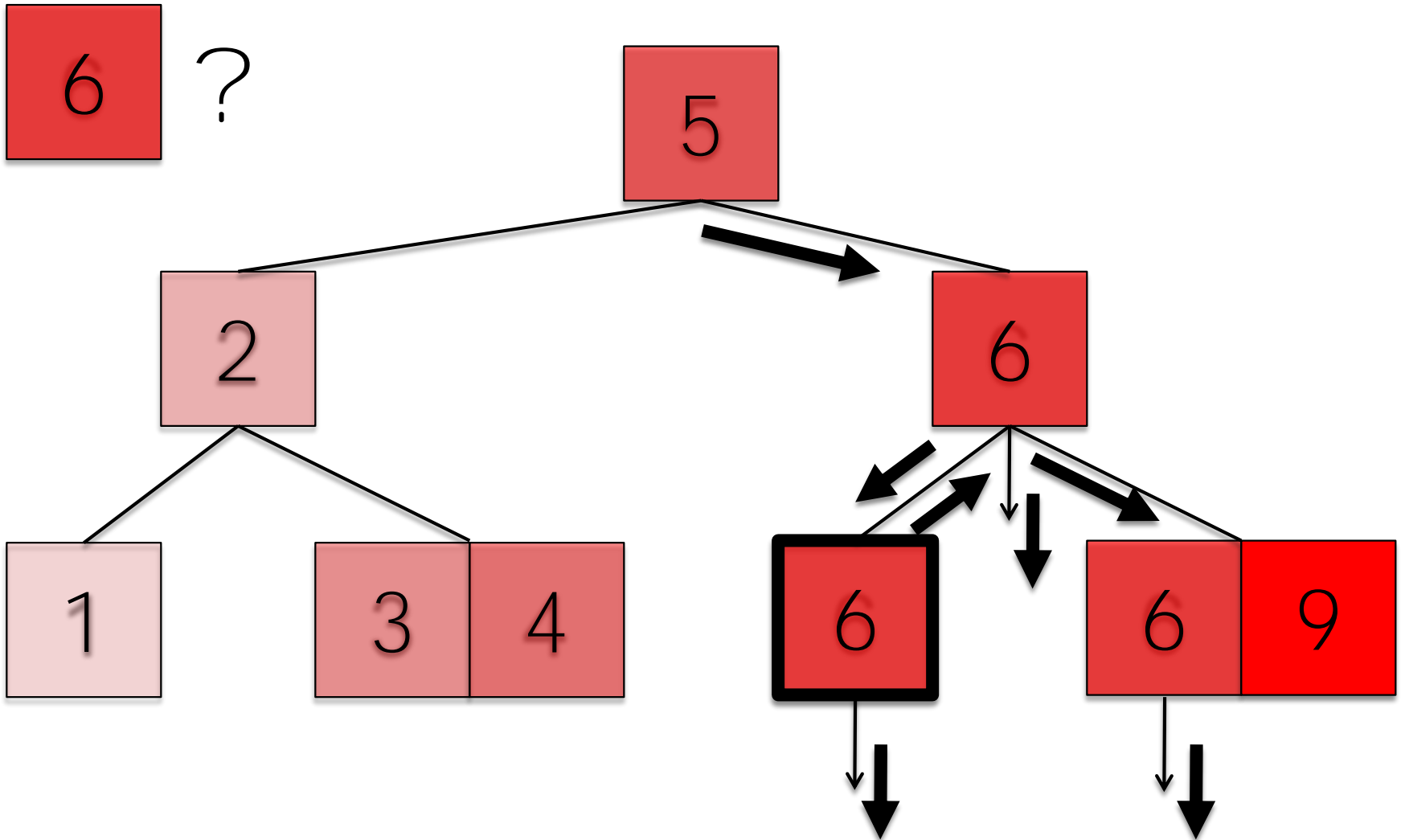
- `db.c.find({x:6})`
- Index `{x:1}`

Equality Match

6 ?



Equality Match



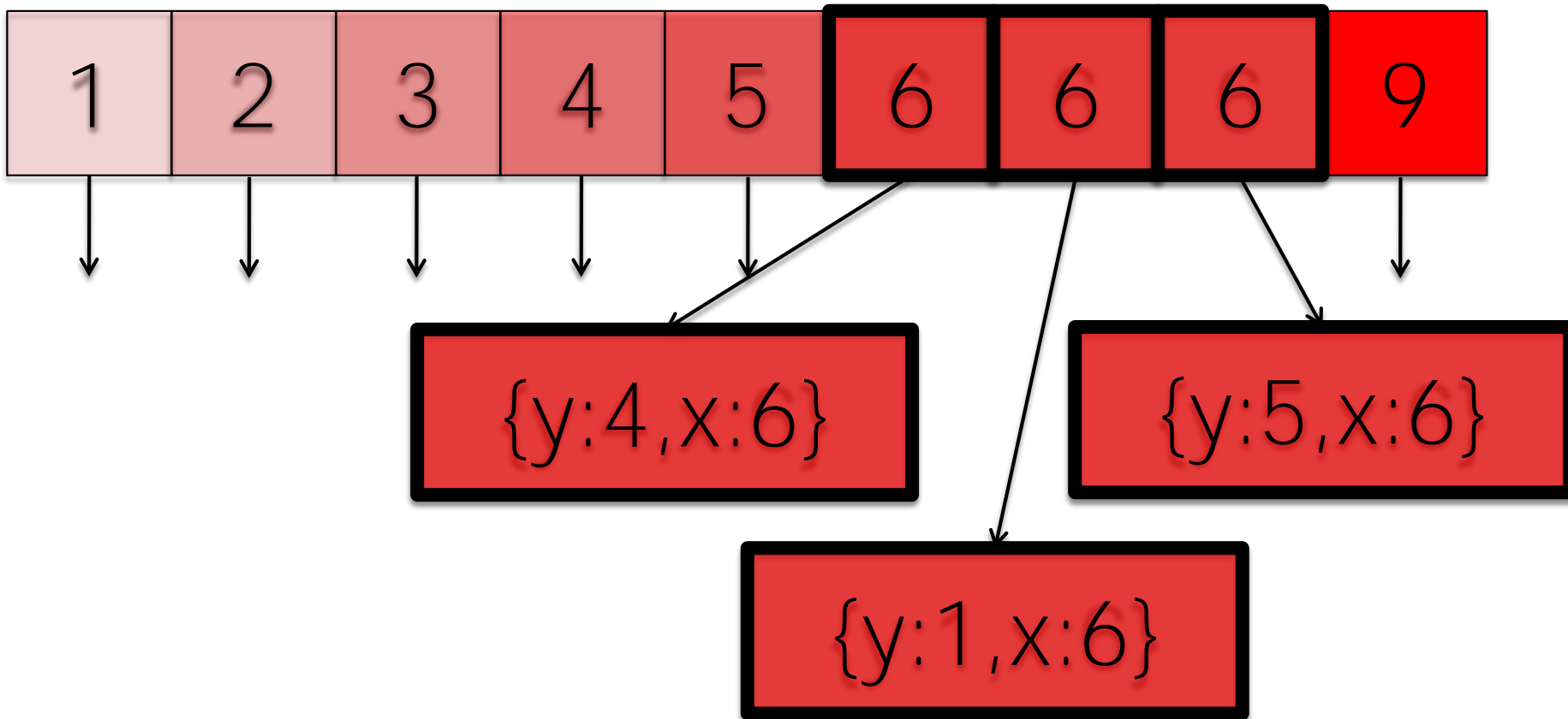
Full Document Matcher

- ❑ `db.c.find({x:6,y:1})`
- ❑ Index `{x:1}`

Full Document Matcher

6

?

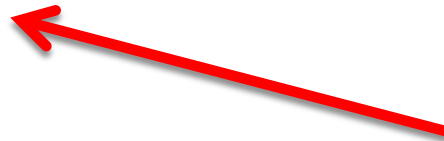


Full Document Matcher

"nscanned" : 3,

"nscannedObjects" : 3,

"n" : 1,

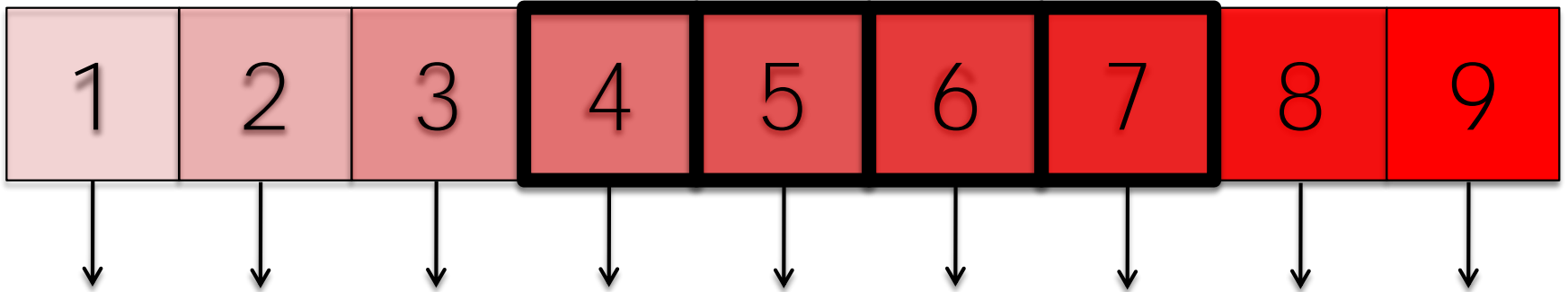
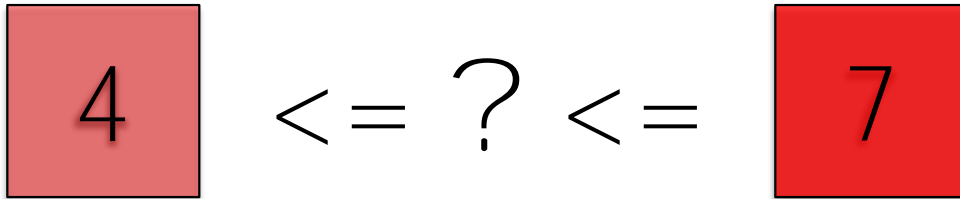


Documents for all matching keys scanned, but only one document matched on non index keys.

Range Match

- `db.c.find({x:{$gte:4,$lte:7}})`
- `Index {x:1}`

Range Match



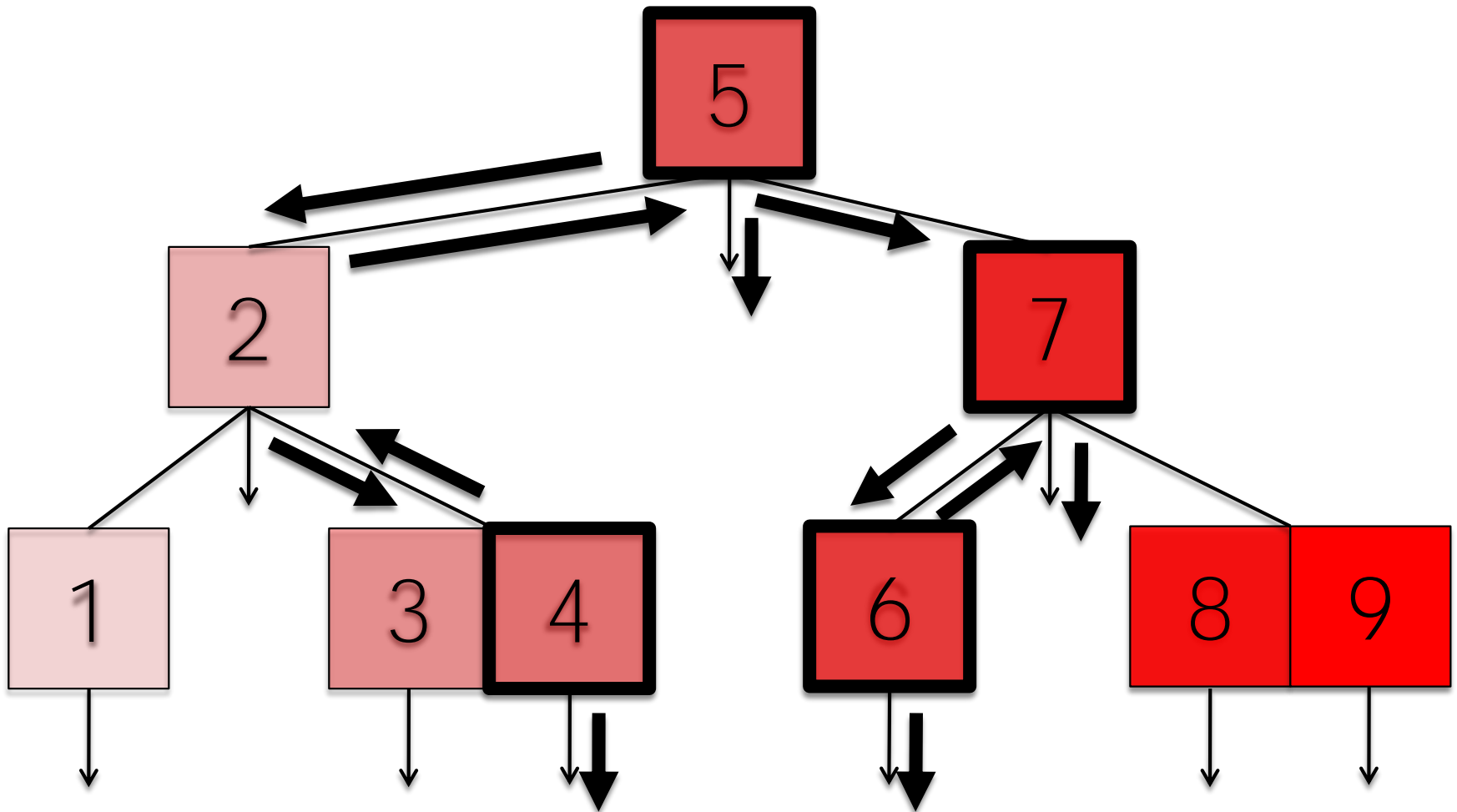
Range Match

"nscanned" : 4,

"nscannedObjects" : 4,

"n" : 4,

Range Match

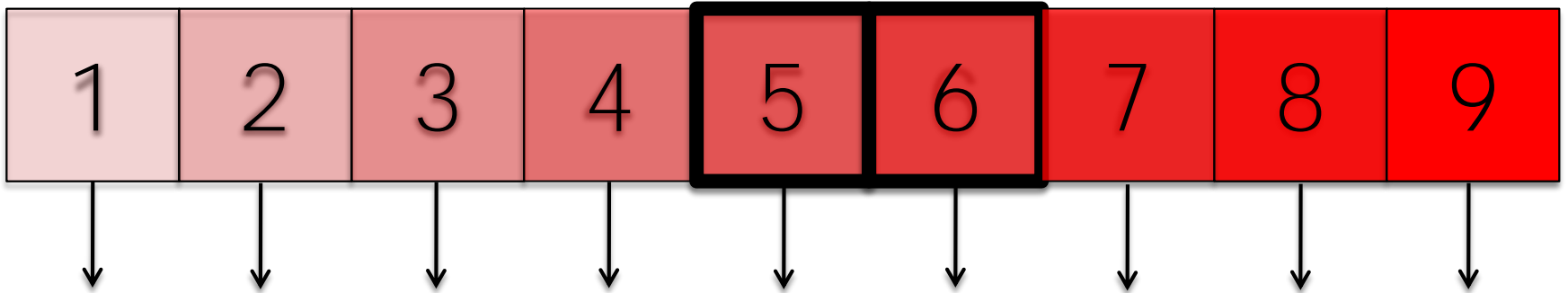


Exclusive Range Match

- `db.c.find({x:{$gt:4,$lt:7}})`
- `Index {x:1}`


Exclusive Range Match

$$\boxed{4} < ? < \boxed{7}$$



Exclusive Range Match

```
"indexBounds" : {  
  "x" : [  
    [  
      4,  
      7  
    ]  
  ]  
}
```



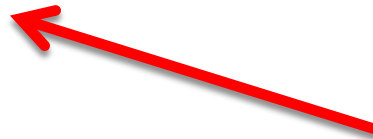
Explain doesn't indicate that the range is exclusive.

Exclusive Range Match

"nscanned" : 2,

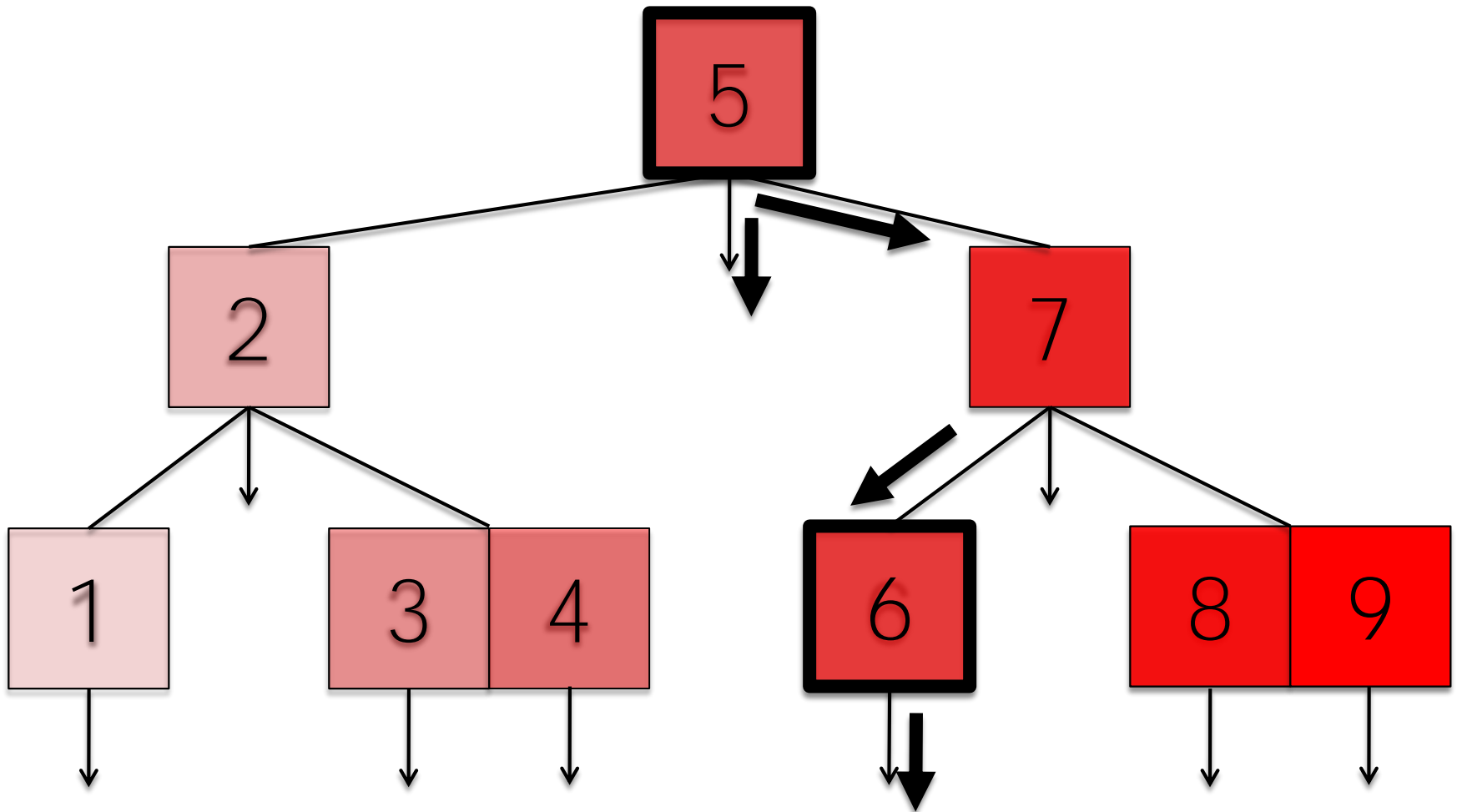
"nscannedObjects" : 2,

"n" : 2,



But index keys matching the range bounds are not scanned because the bounds are exclusive.

Exclusive Range Match



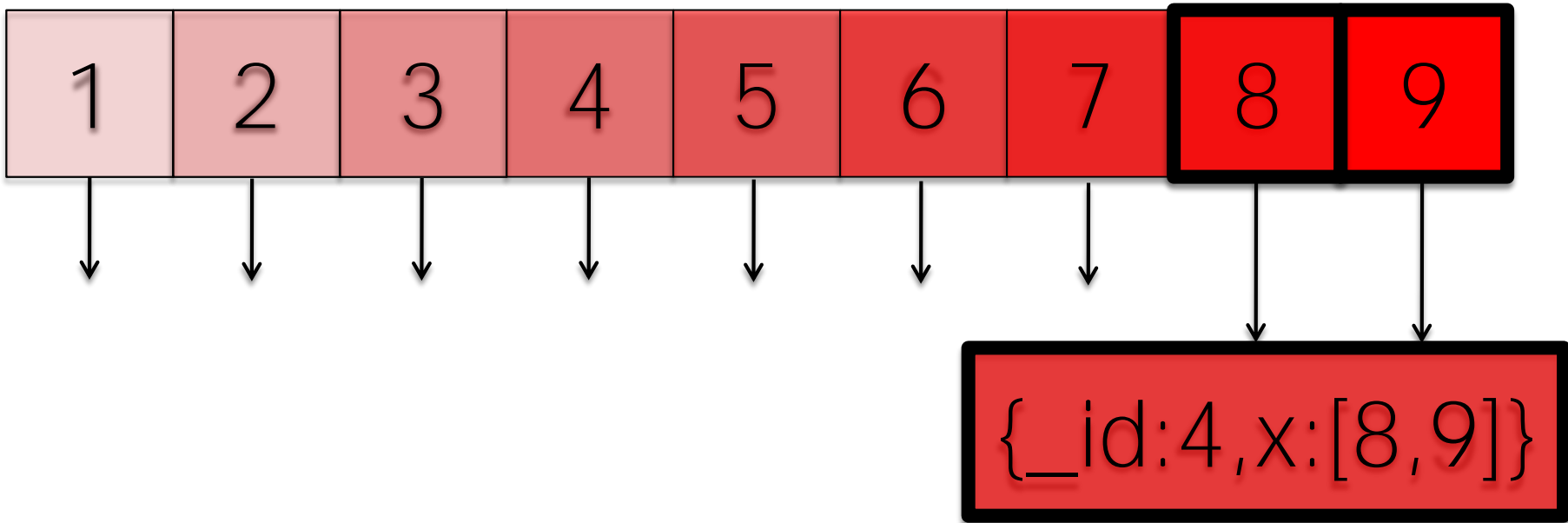
Multikeys

- `db.c.find({x:{$gt:7}})`
- Index `{x:1}`

Multikkeys

? >

7

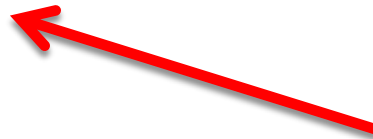


Multikeys

"nscanned" : 2,

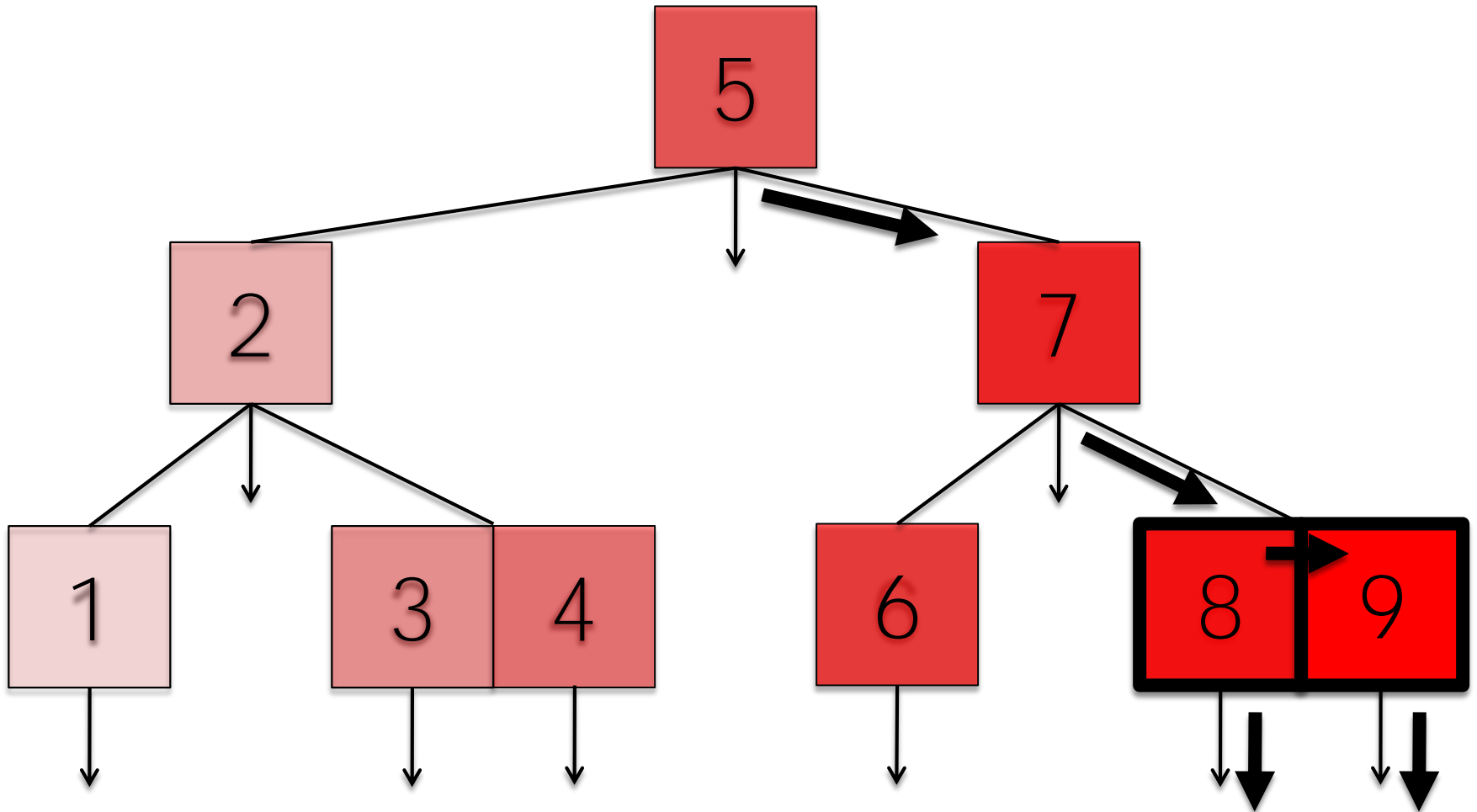
"nscannedObjects" : 2,

"n" : 1,



All keys in valid range are scanned, but the matcher rejects duplicate documents making $n == 1$.

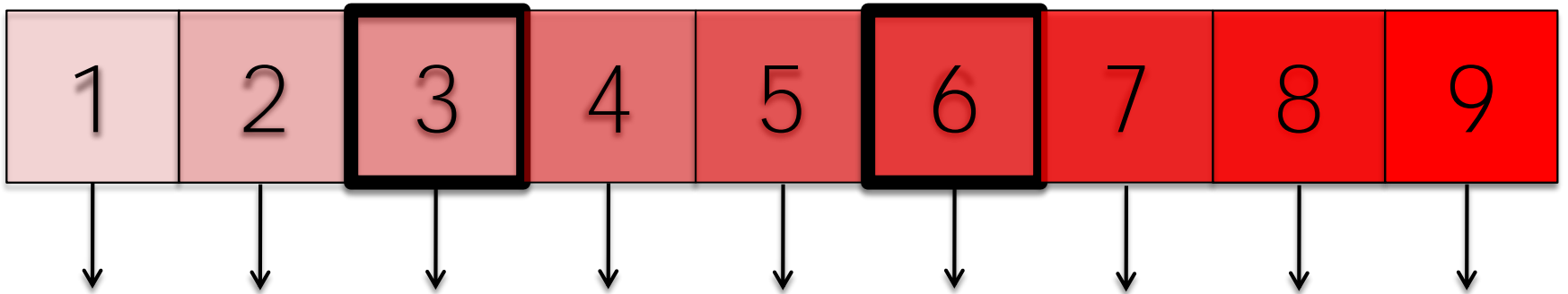
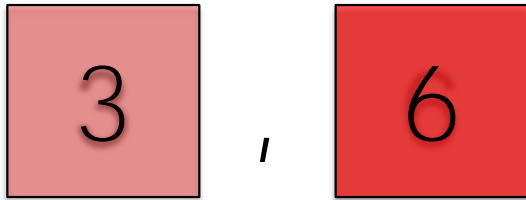
Multikkeys



Set Match

- `db.c.find({x:{$in:[3,6]}})`
- Index {x:1}

Set Match

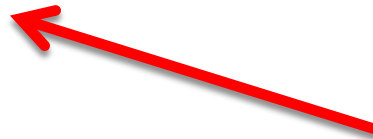


Set Match

"nscanned" : 3,

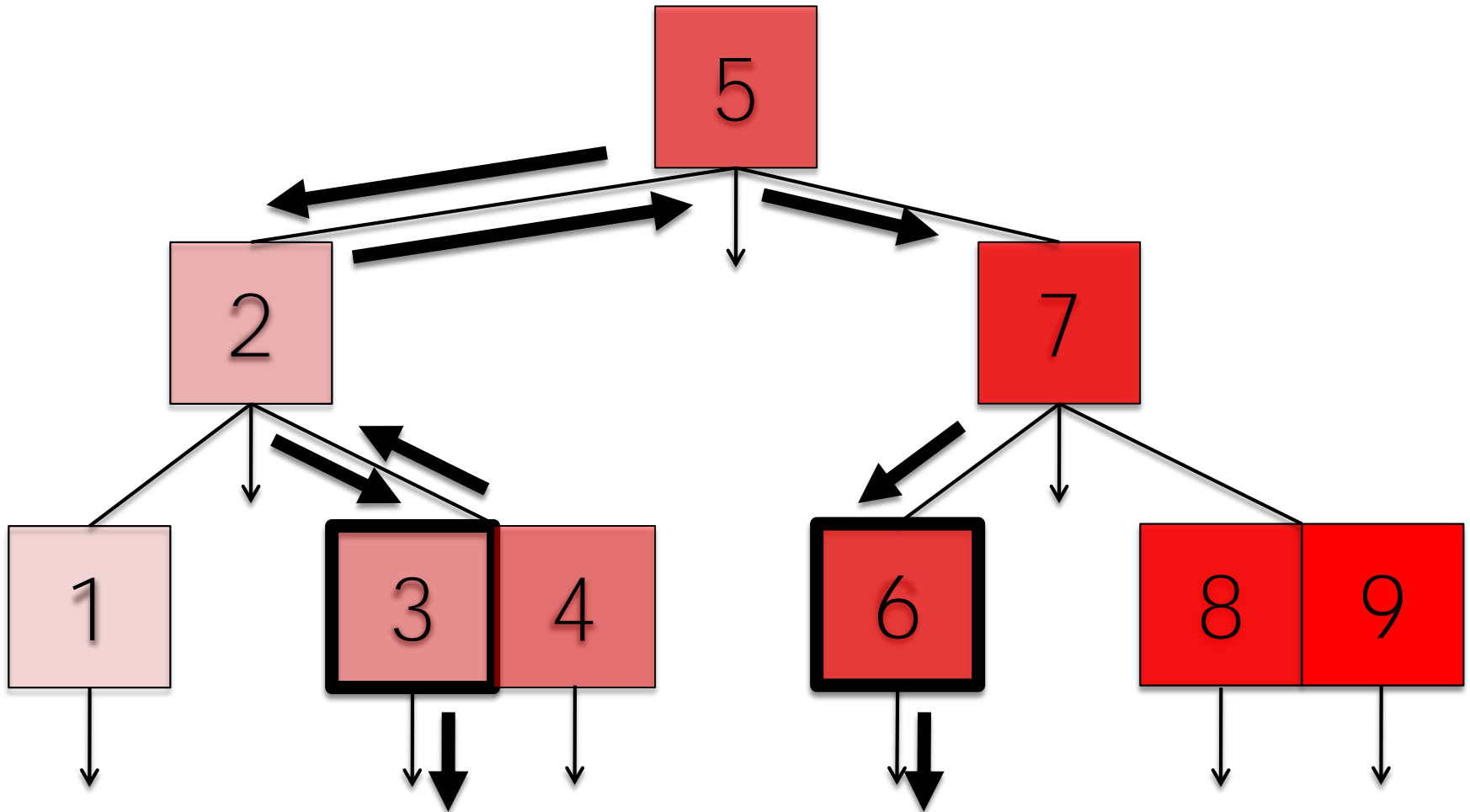
"nscannedObjects" : 2,

"n" : 2,



Why is nscanned 3?
This is an
algorithmic detail
we'll discuss more
later, but when there
are disjoint ranges
for a key nscanned
may be higher than
the number of
matching keys.

Set Match

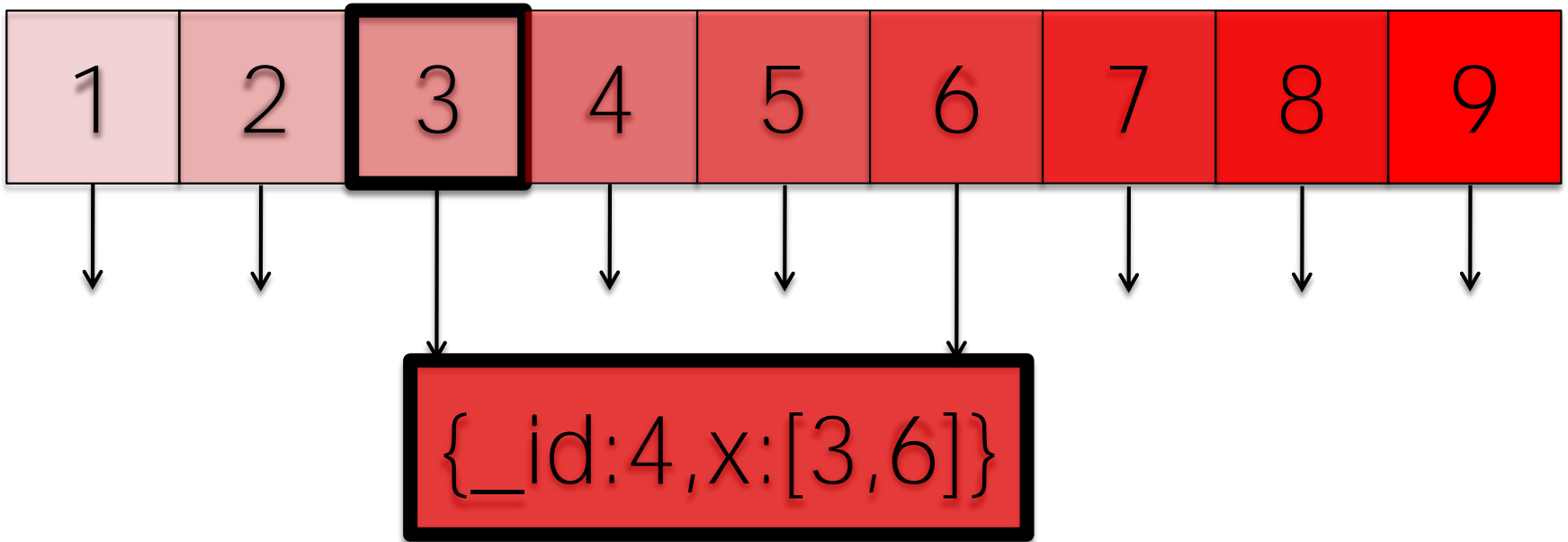


All Match

- ❑ `db.c.find({x:{$all:[3,6]}})`
- ❑ Index {x:1}

All Match

3 ?



All Match

```
"indexBounds" : {  
  "x" : [  
    [  
      3,  
      3  
    ]  
  ]  
}
```

The first entry in the \$all match array is always used for index bounds. Note this may not be the least numerous indexed value in the \$all array.

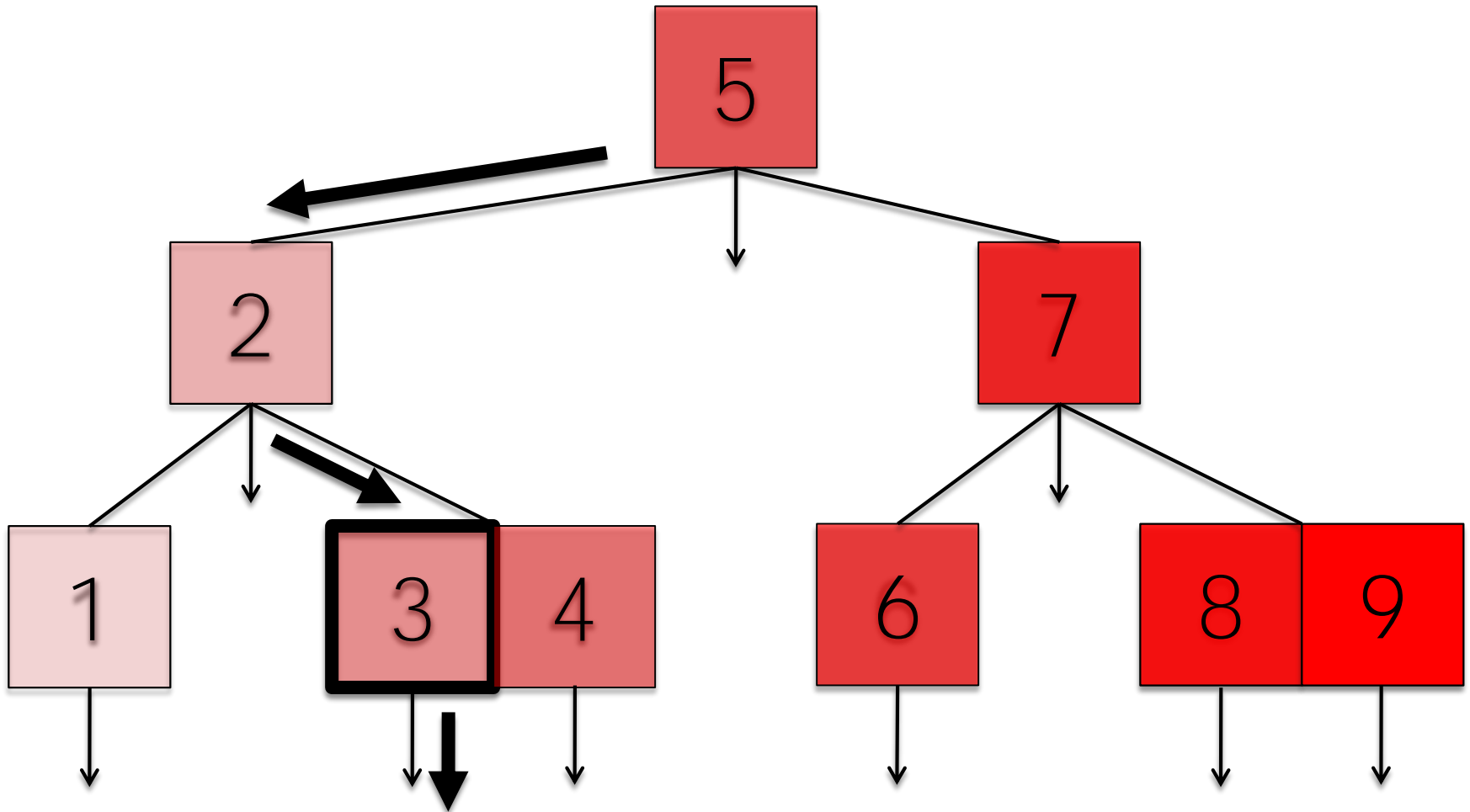
All Match

"nscanned" : 1,

"nscannedObjects" : 1,

"n" : 1,

All Match



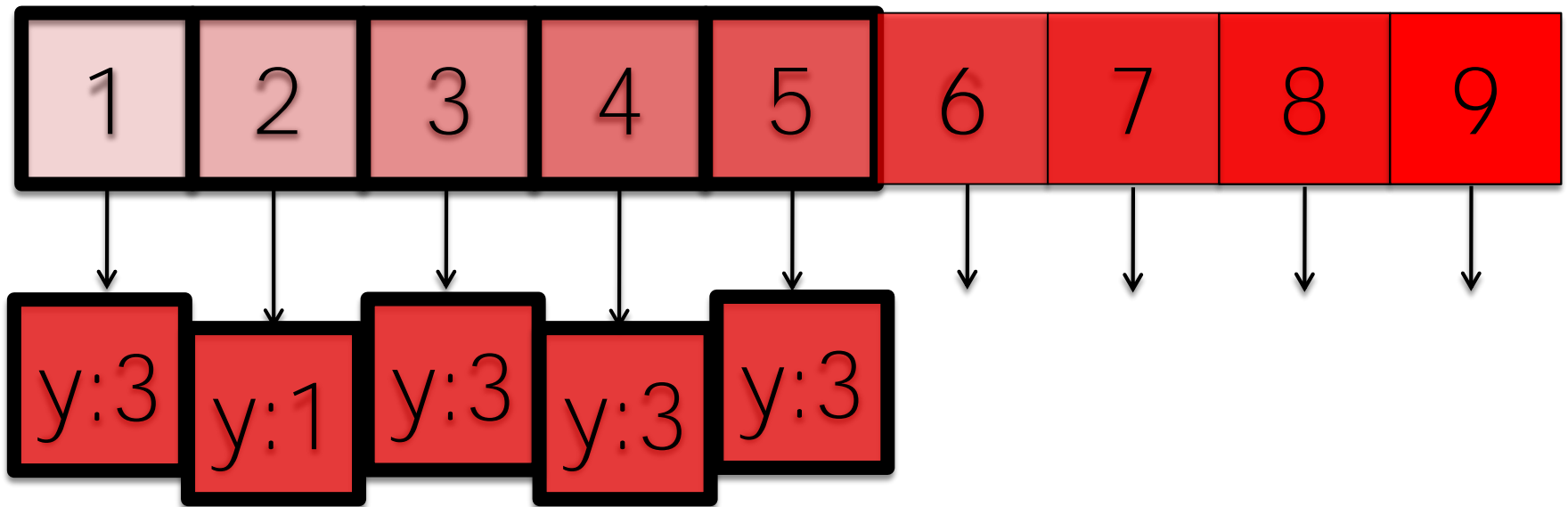
Limit

- `db.c.find({x:{<6},y:3}).limit(3)`
- `Index {x:1}`

Limit

? <

6

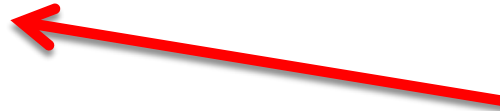


Limit

"nscanned" : 4,

"nscannedObjects" : 4,

"n" : 3,



Scan until three
matches are found,
then stop.

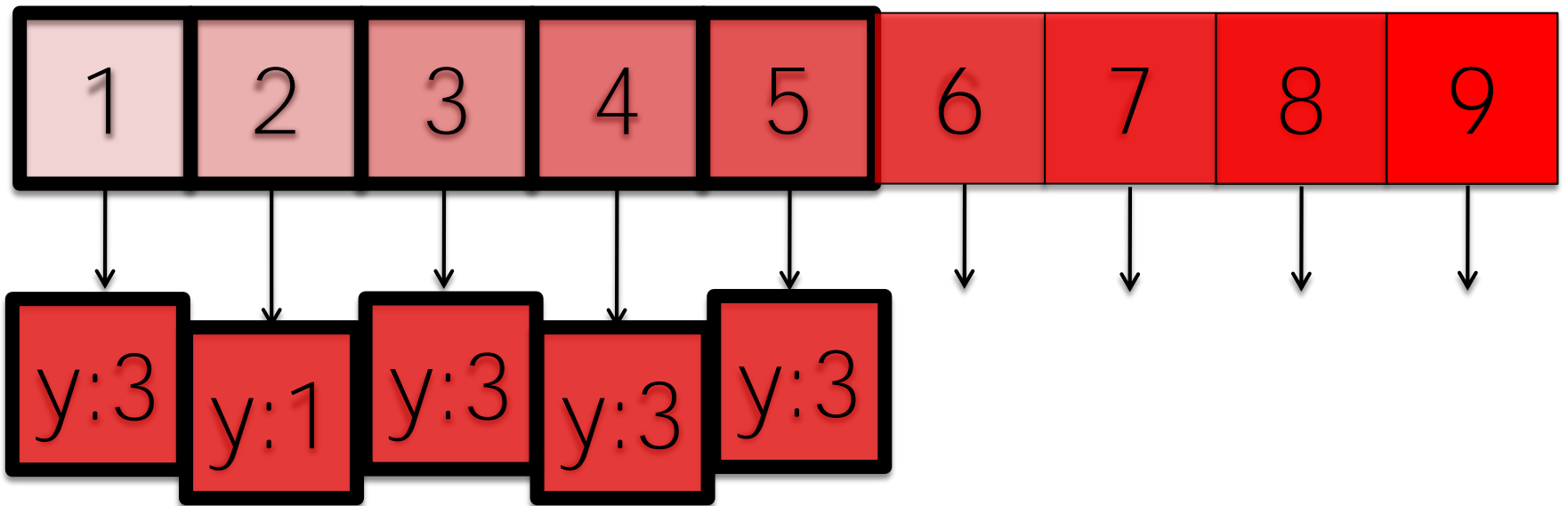
Skip

- `db.c.find({x:{$lt:6},y:3}).skip(3)`
- `Index {x:1}`

Skip

? <

6

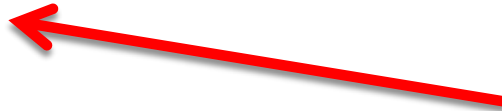


Skip

"nscanned" : 5,

"nscannedObjects" : 5,

"n" : 1,



All skipped
documents are
scanned.

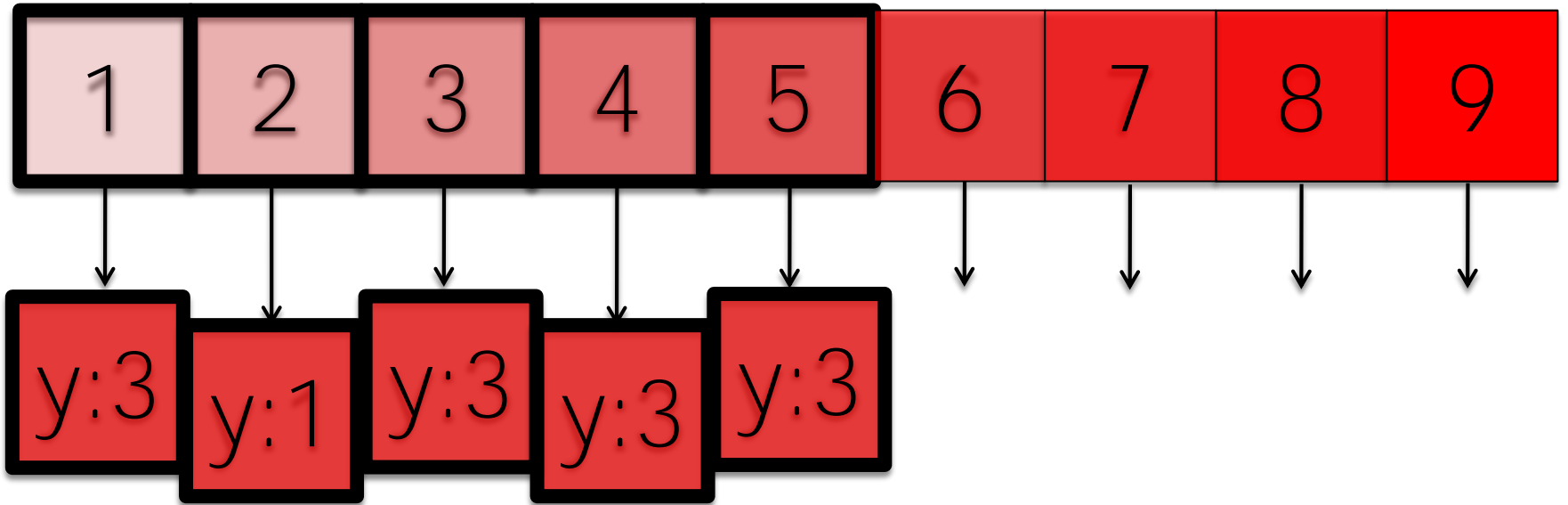
Sort

- `db.c.find({x:{$lt:6}}).sort({x:1})`
- Index {x:1}

Sort

? <

6



Sort

```
"cursor" : "BtreeCursor x_1",
```

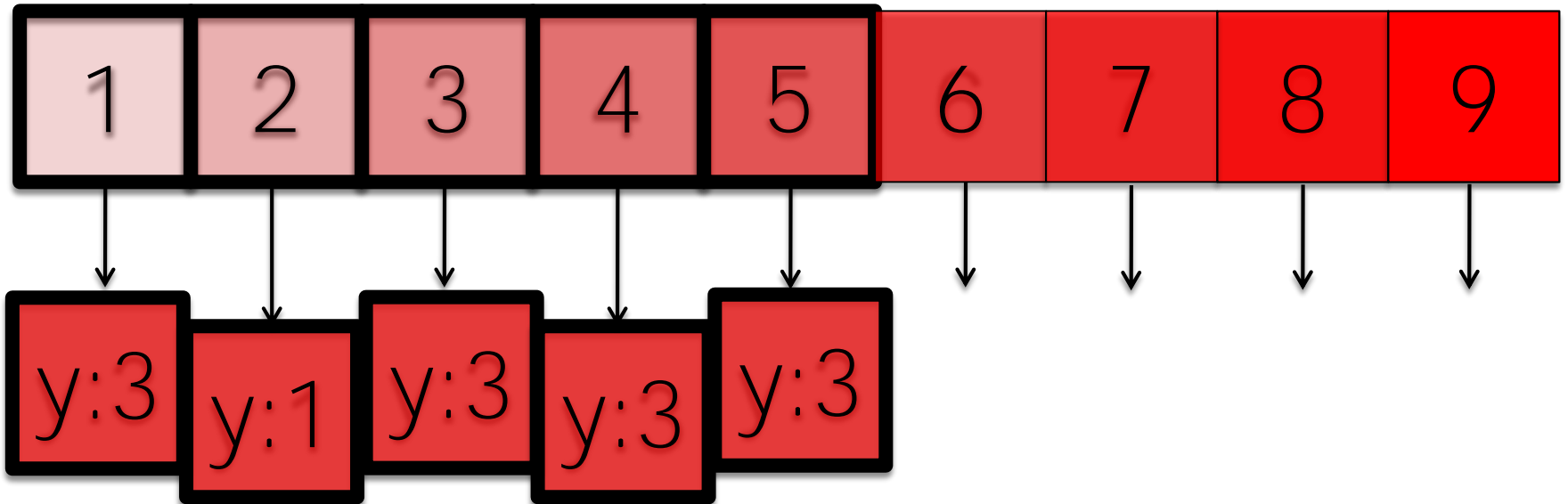
Sort

- `db.c.find({x:{$lt:6}}).sort({y:1})`
- Index `{x:1}`

Sort

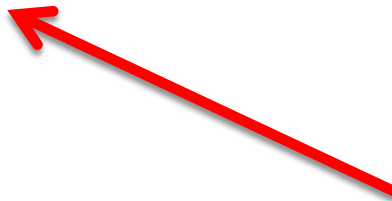
? <

6



Sort

```
"cursor" : "BtreeCursor x_1",  
"nscanned" : 5,  
"nscannedObjects" : 5,  
"n" : 4,  
"scanAndOrder" : true,
```



Results are sorted on the fly to match requested order. The scanAndOrder field is only printed when its value is true.

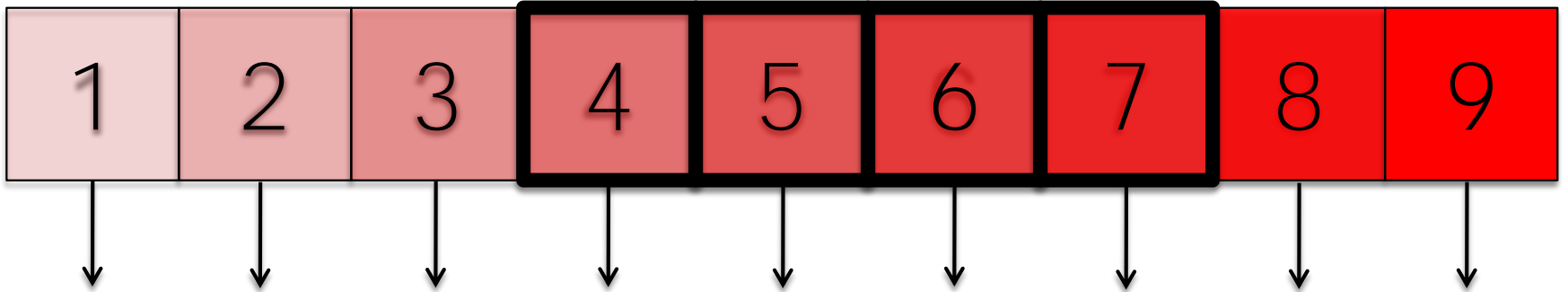
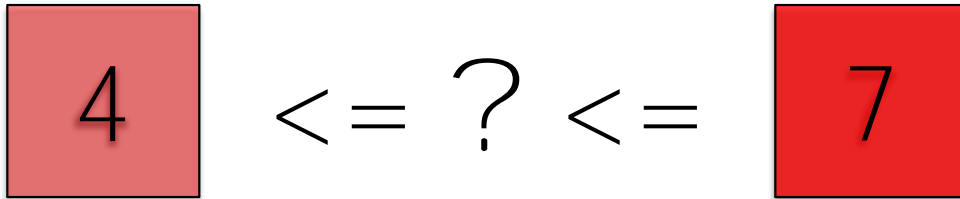
Sort and scanAndOrder

- With “scanAndOrder” sort, all documents must be touched even if there is a limit spec.
- With scanAndOrder, sorting is performed in memory and the memory footprint is constrained by the limit spec if present.

Count

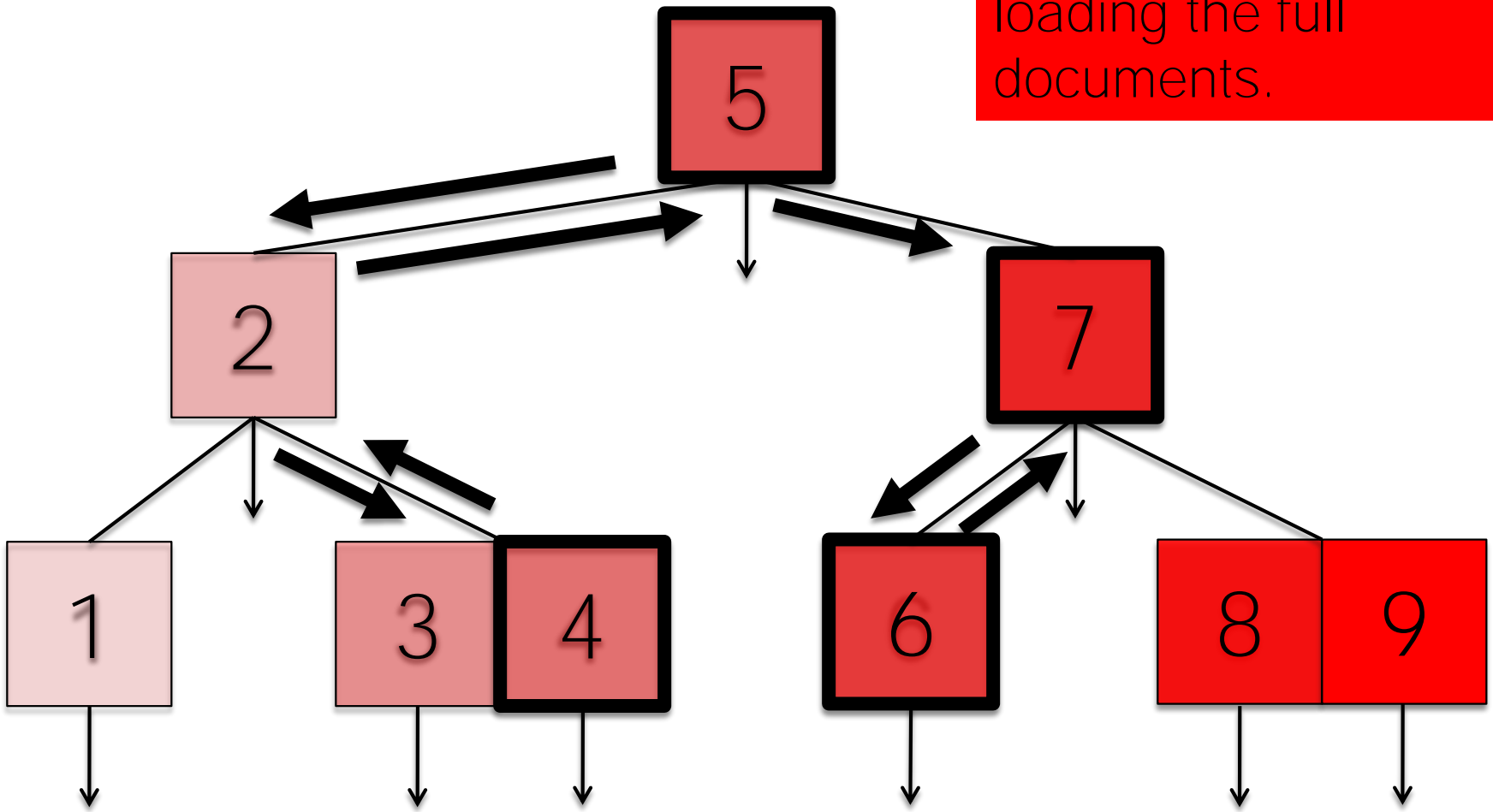
- `db.c.count({x:{$gte:4,$lte:7}})`
- `Index {x:1}`

Count



Count

We're just counting keys here, not loading the full documents.

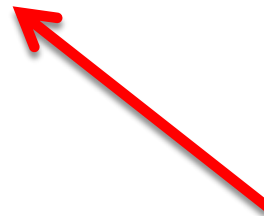


Count

- With some operators the full document must be checked. Some of these cases:
 - \$all
 - \$size
 - array match
 - Negation - \$ne, \$nin, \$not, etc.
 - With current semantics, all multikey elements must match negation constraints
- Multikey de duplication works without loading full document

Covered Indexes

- `db.c.find({x:6}, {x:1,_id:0})`
- Index `{x:1}`

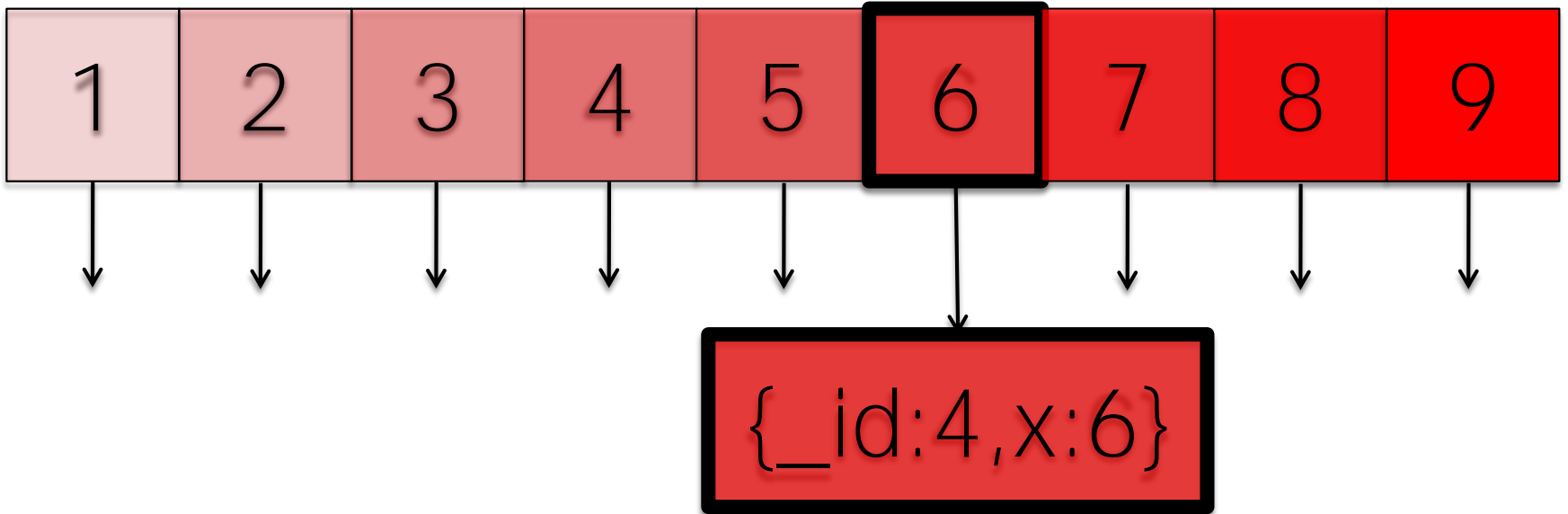


Id would be returned **by default, but isn't** in the index so we need to exclude to return only indexed fields.

Covered Indexes

6

?



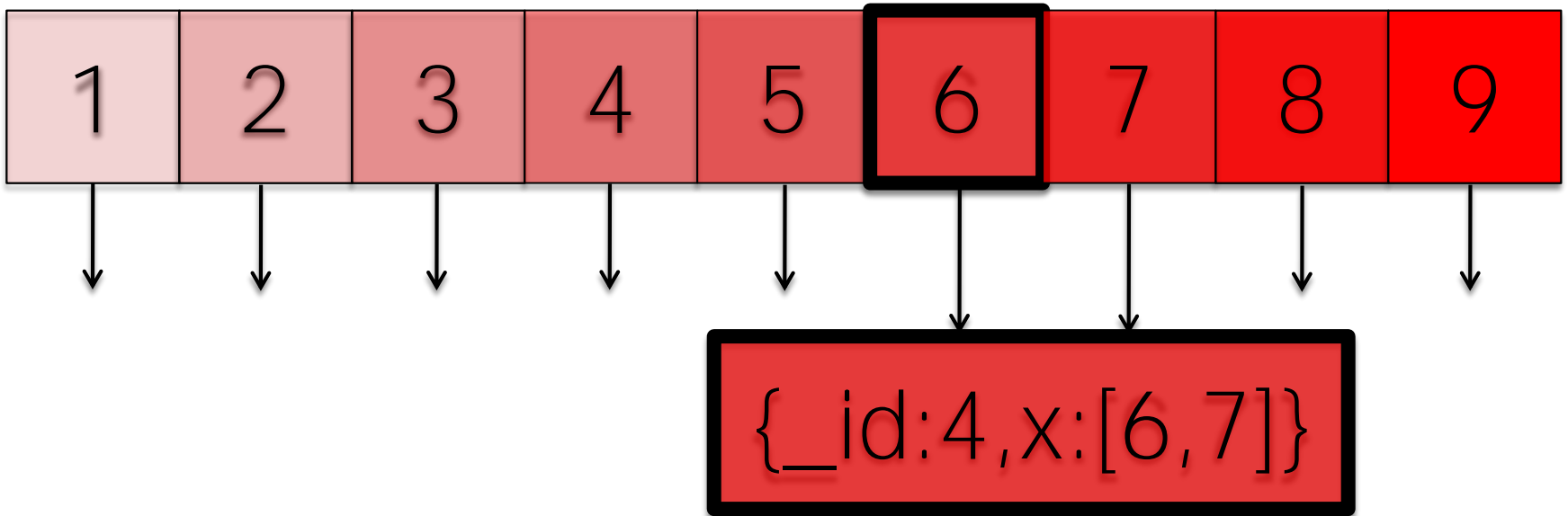
Covered Indexes

"isMultiKey" : false,

"indexOnly" : true,

Covered Indexes

6 ?



Covered Indexes

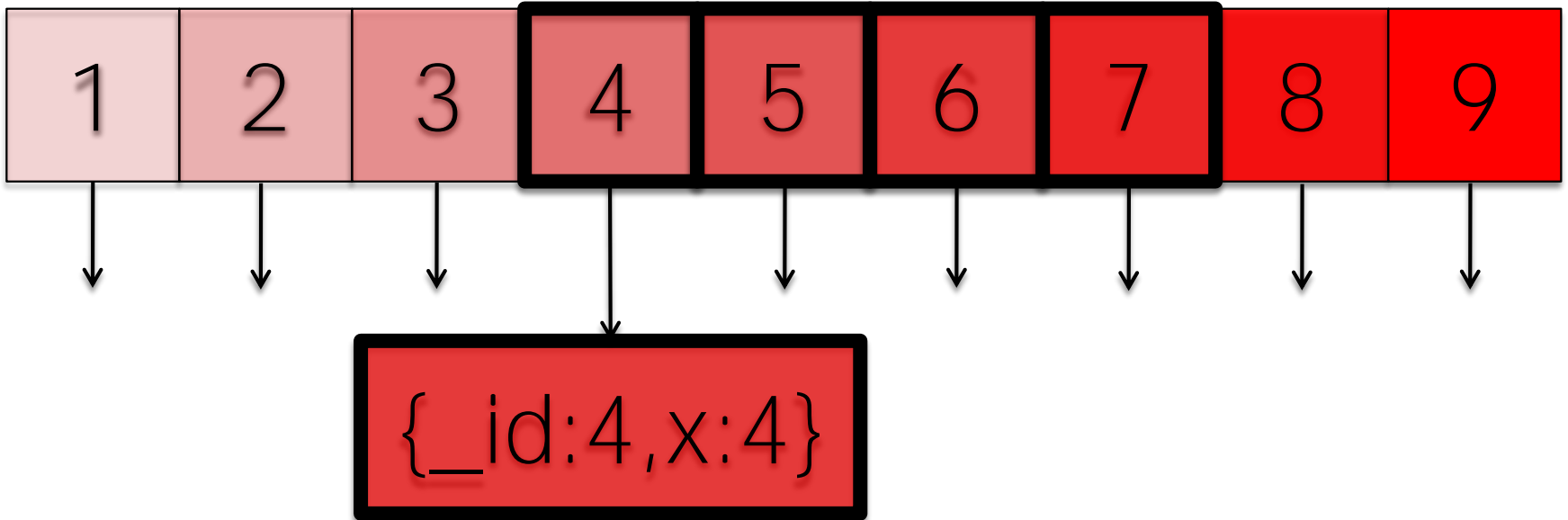
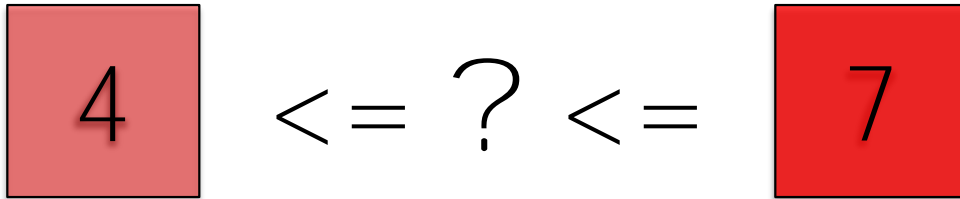
"isMultiKey" : true,
"indexOnly" : false,

Currently we set isMultiKey to true the first time we save a doc where the field is a multikey array. But when all multikey docs **are removed we don't** reset isMultiKey. This can be improved.

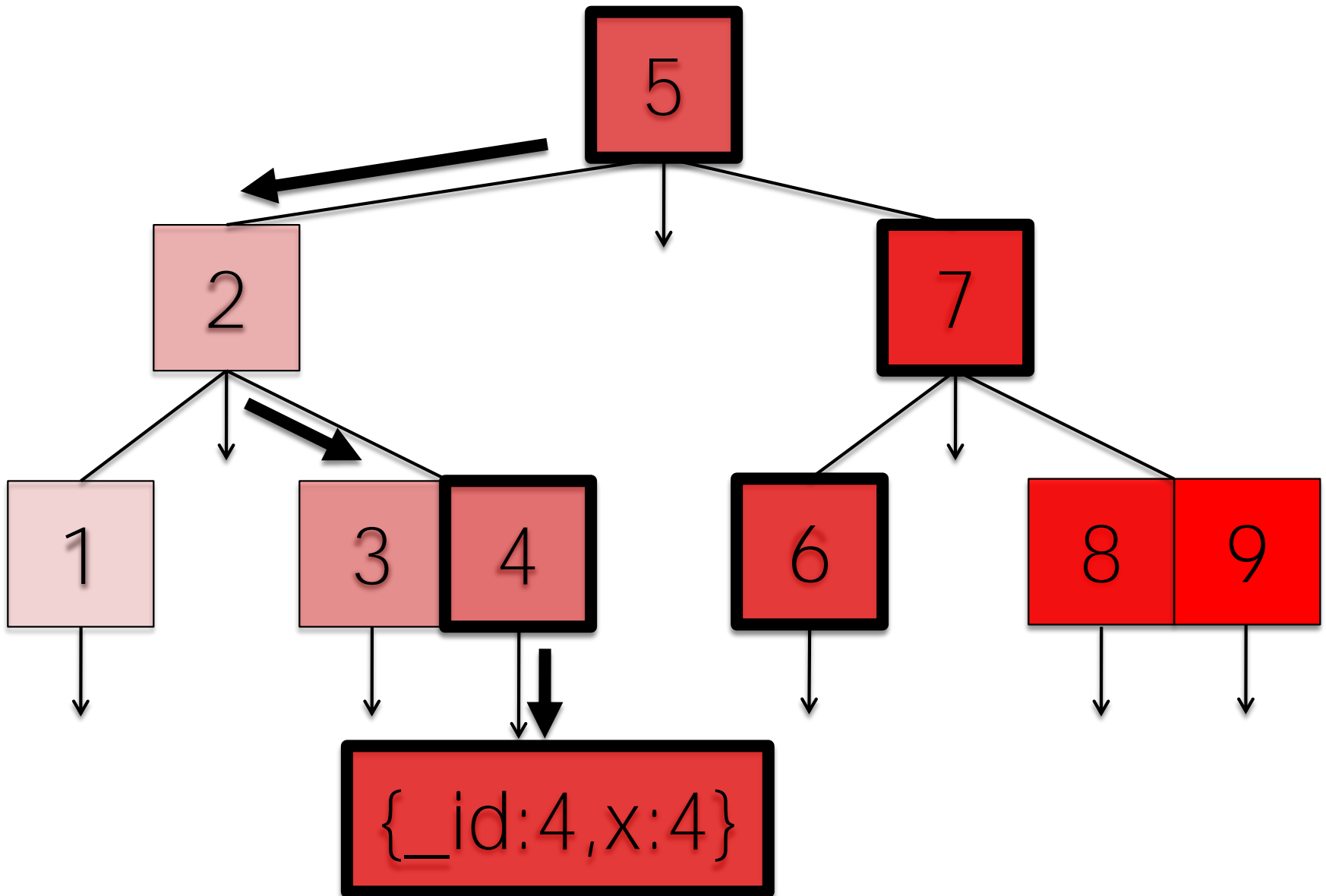
Update

- `db.c.find({x:{$gte:4,$lte:7}}, {$set:{x:2}})`
- Index {x:1}

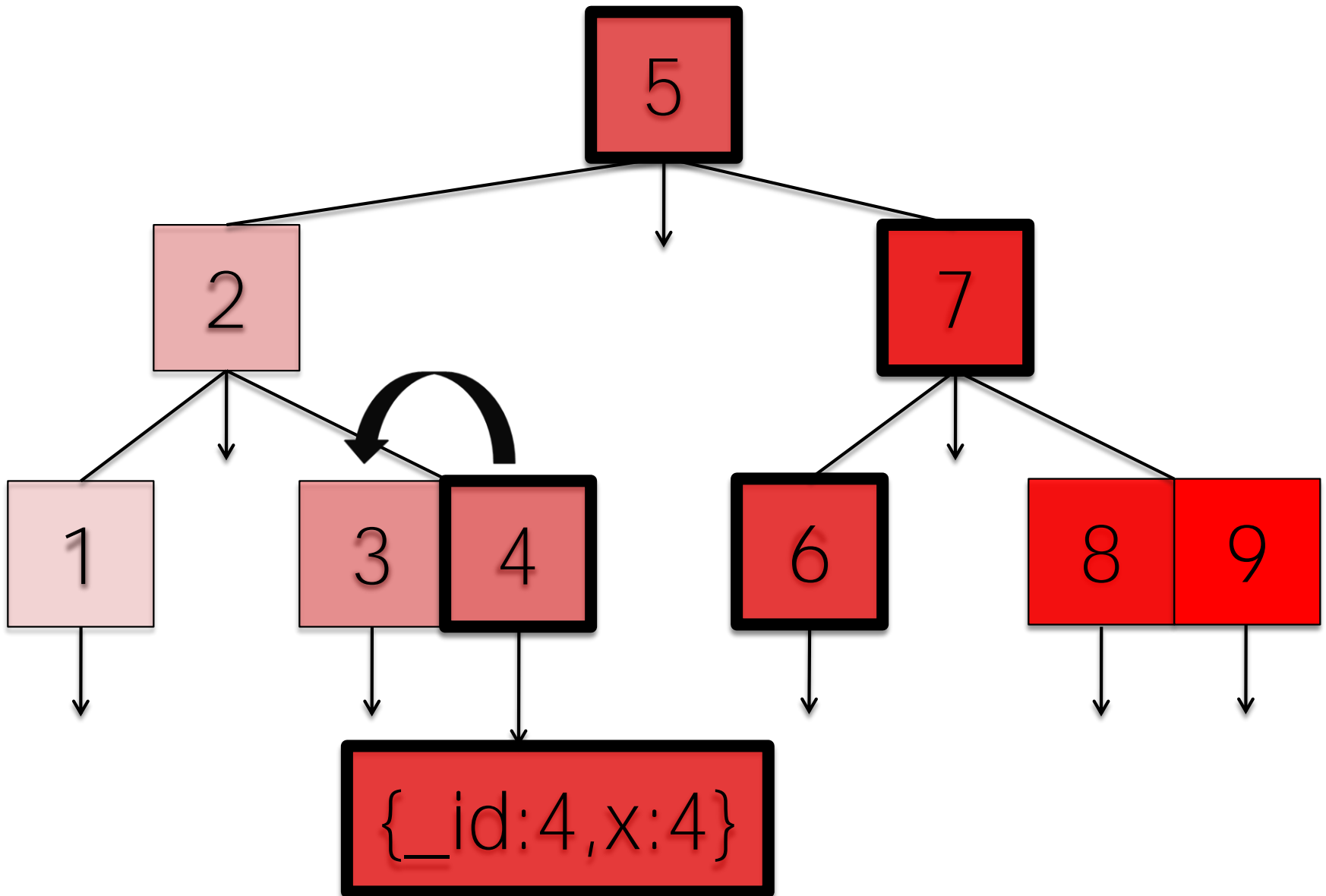
Update



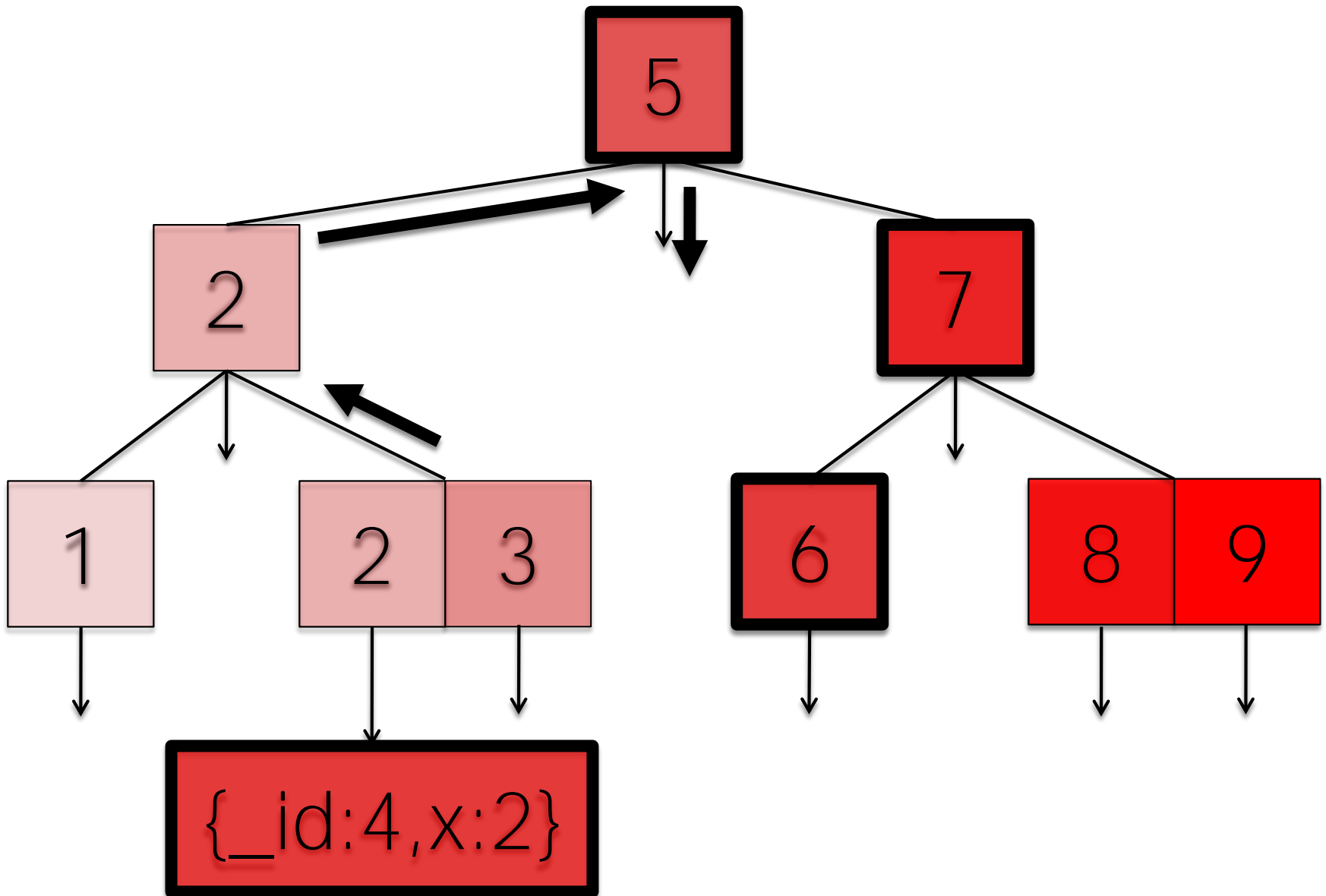
Update



Update



Update



Update

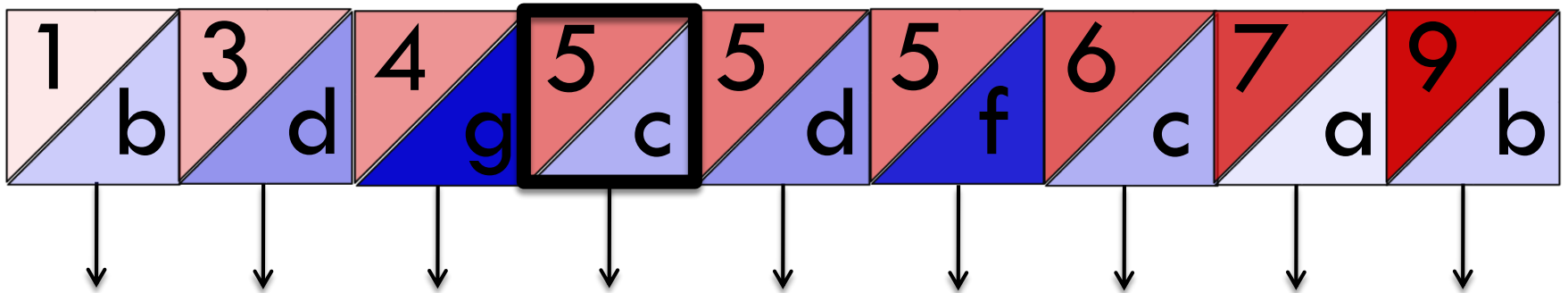
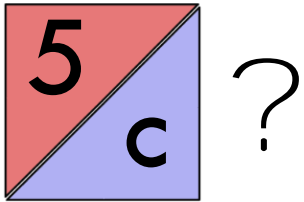
- We track the set of documents that have been updated in the course of the current operation so they are only updated once.

Compound Key Index Bounds

Two Equality Bounds

- `db.c.find({x:5,y:'c'})`
- Index `{x:1,y:1}`

Two Equality Bounds



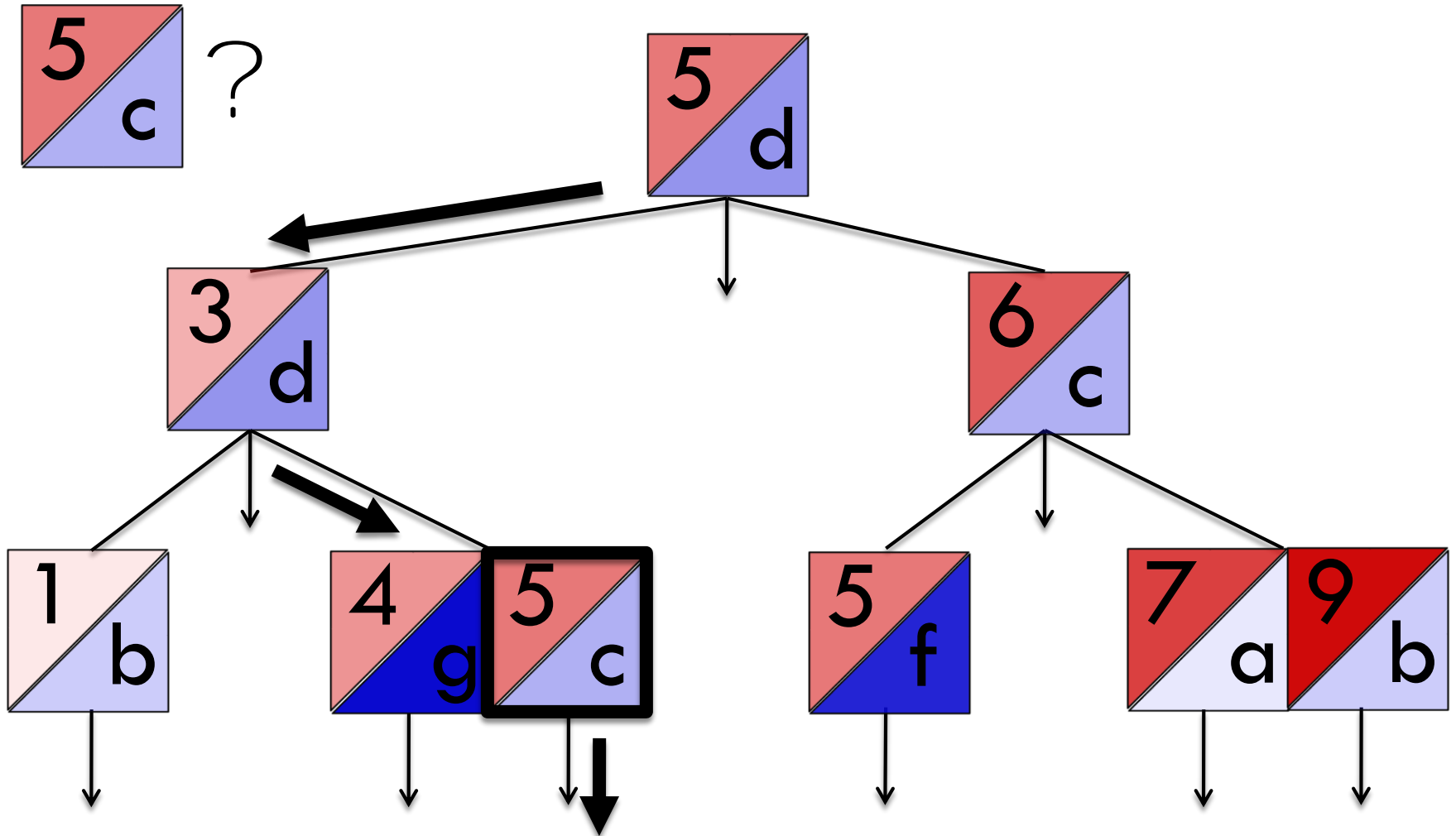
Two Equality Bounds

"nscanned" : 1,

"nscannedObjects" : 1,

"n" : 1,

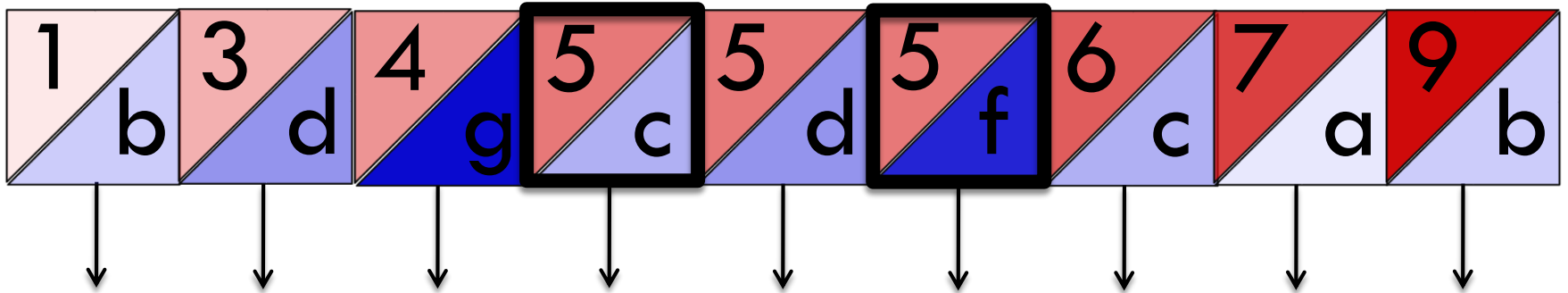
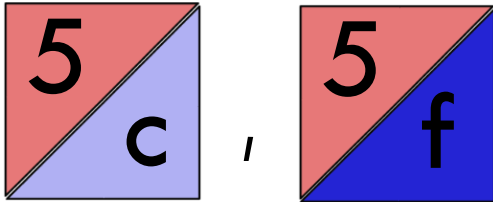
Two Equality Bounds



Equality and Set

- `db.c.find({x:5,y:{$in:['c','f']}})`
- Index `{x:1,y:1}`

Equality and Set



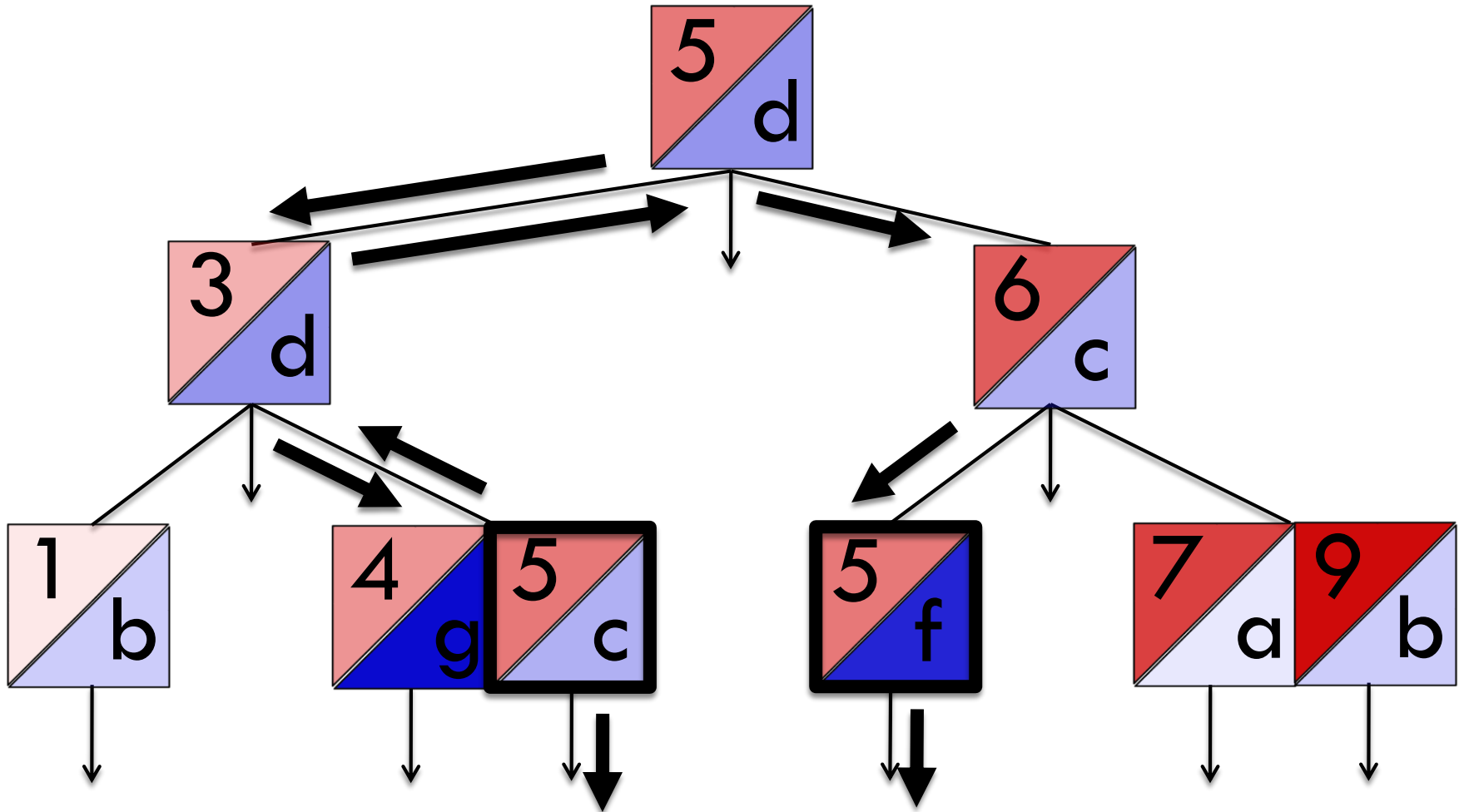
Equality and Set

"nscanned" : 3,

"nscannedObjects" : 2,

"n" : 2,

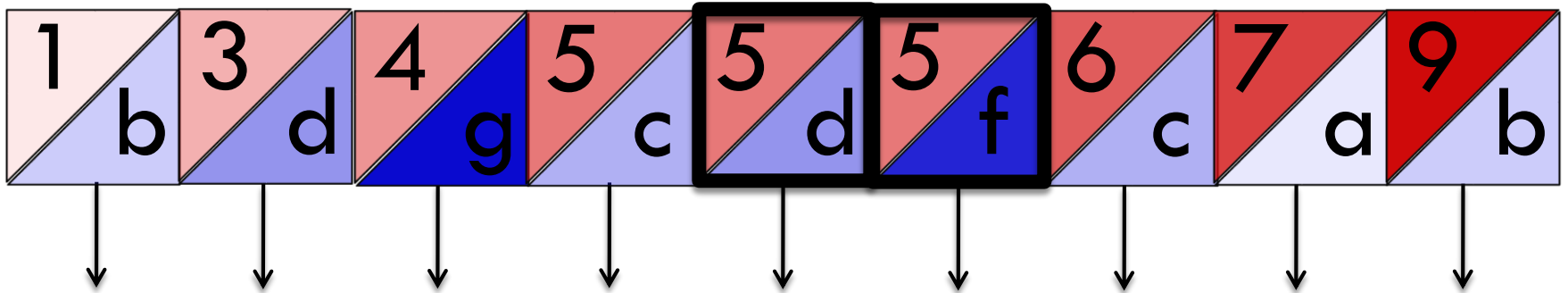
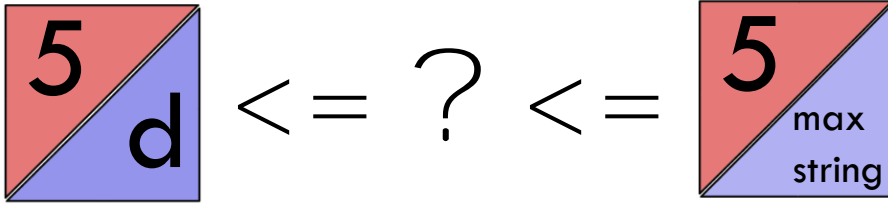
Equality and Set



Equality and Range

- `db.c.find({x:5,y:{$gte:'d'}})`
- `Index {x:1,y:1}`

Equality and Range



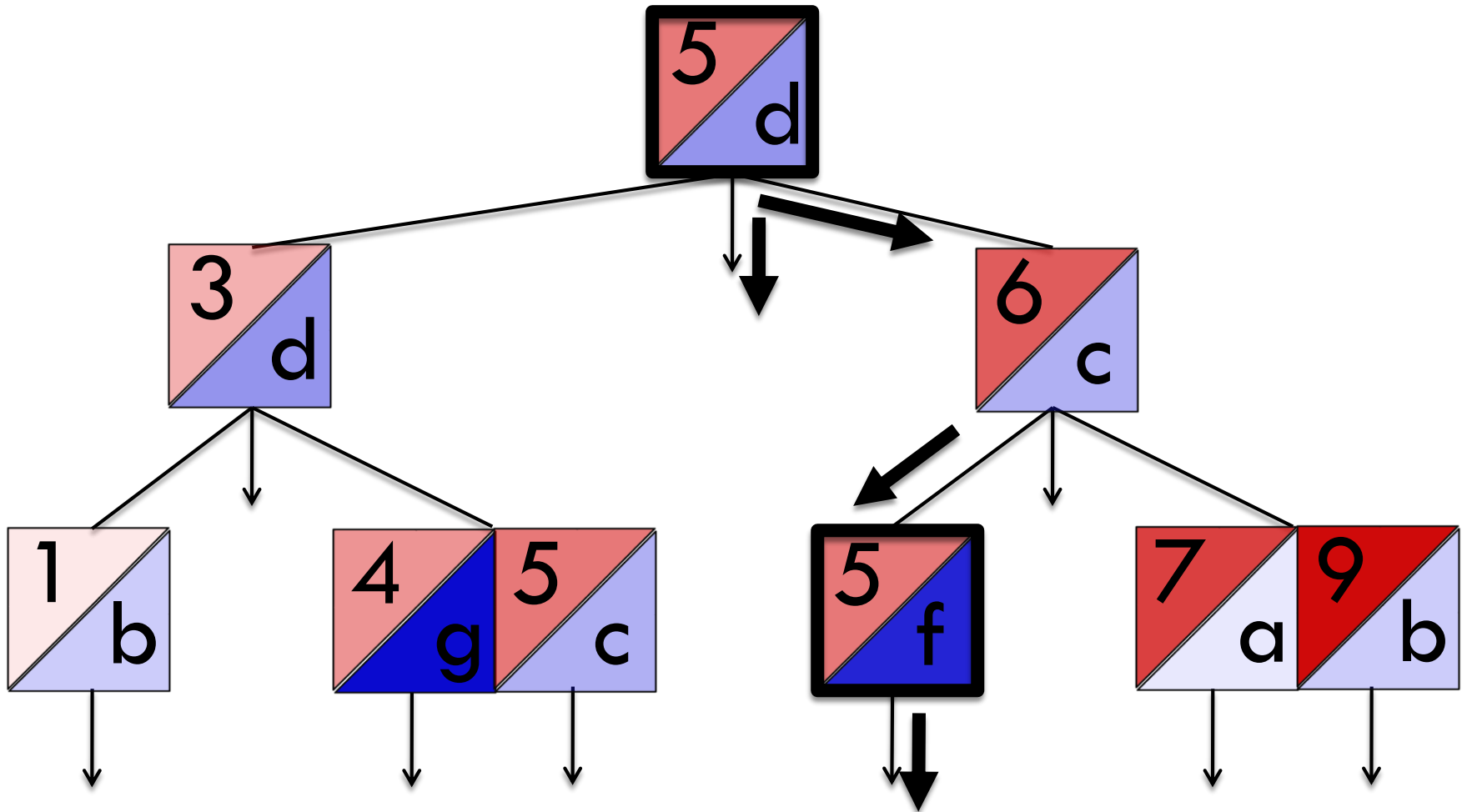
Equality and Range

"nscanned" : 2,

"nscannedObjects" : 2,

"n" : 2,

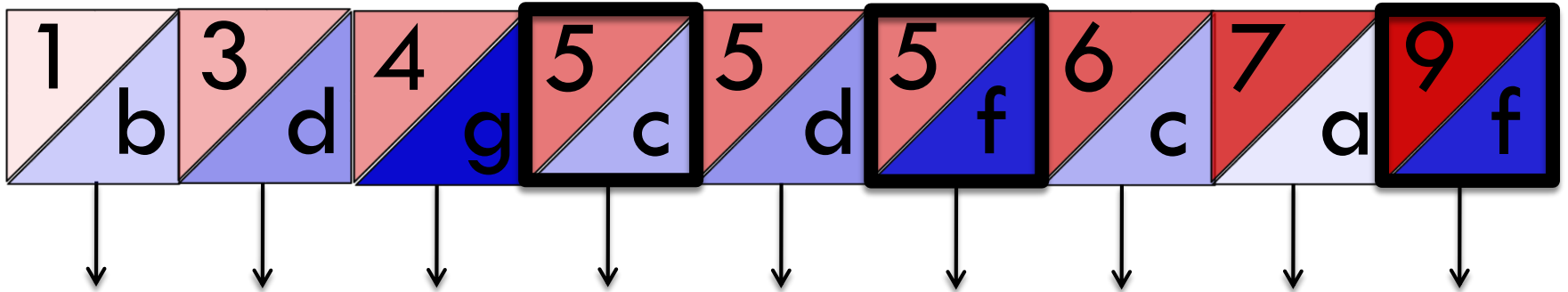
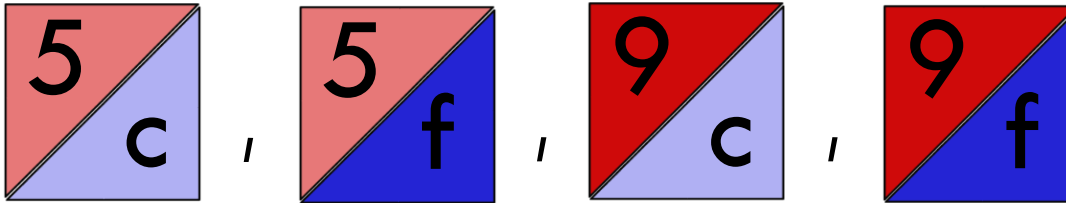
Equality and Range



Two Set Bounds

- `db.c.find({x:{$in:[5,9]},y:{$in:['c','f']}})`
- Index `{x:1,y:1}`

Two Set Bounds



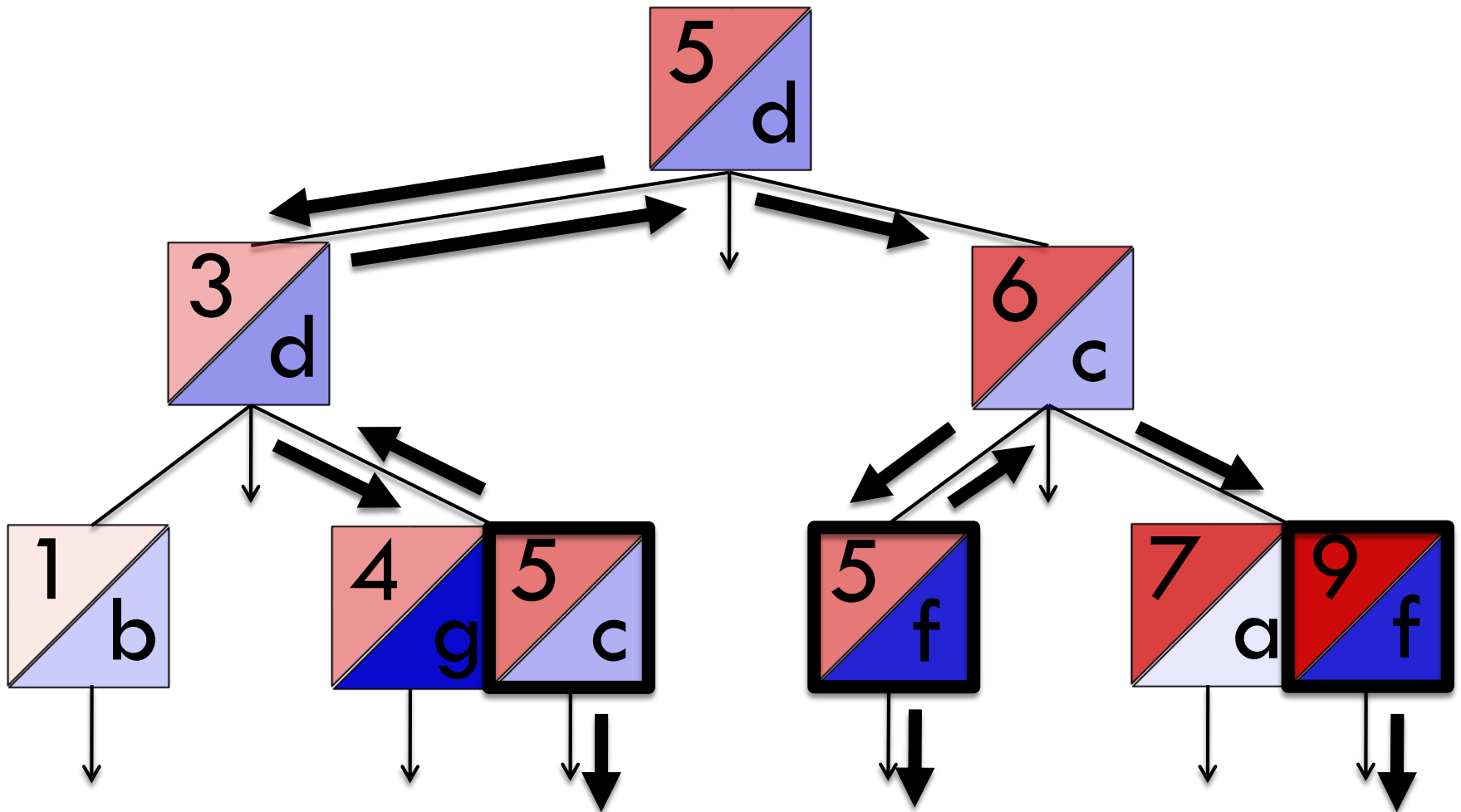
Two Set Bounds

"nscanned" : 5,

"nscannedObjects" : 3,

"n" : 3,

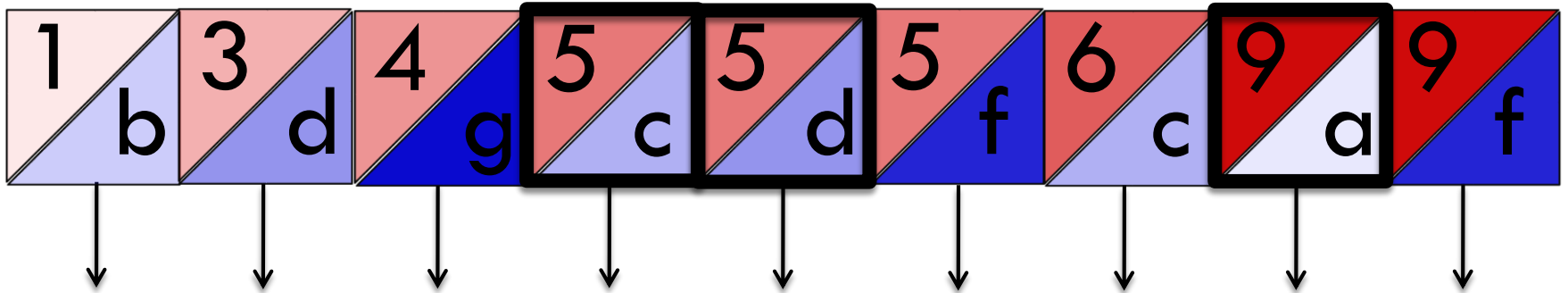
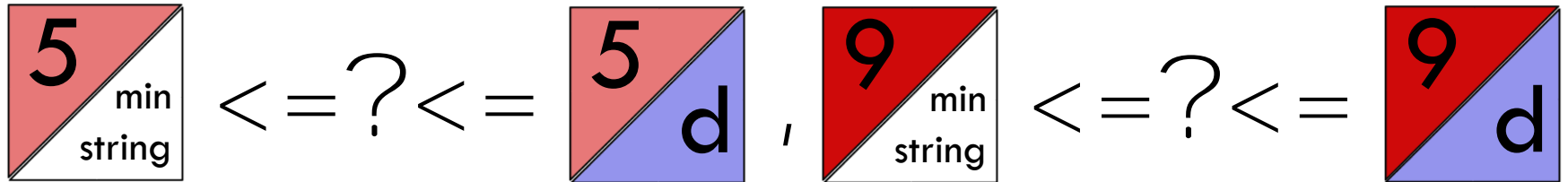
Two Set Bounds



Set and Range

- `db.c.find({x:{$in:[5,9]},y:{$lte:'d'}})`
- Index `{x:1,y:1}`

Set and Range



Set and Range

"nscanned" : 5,

"nscannedObjects" : 3,

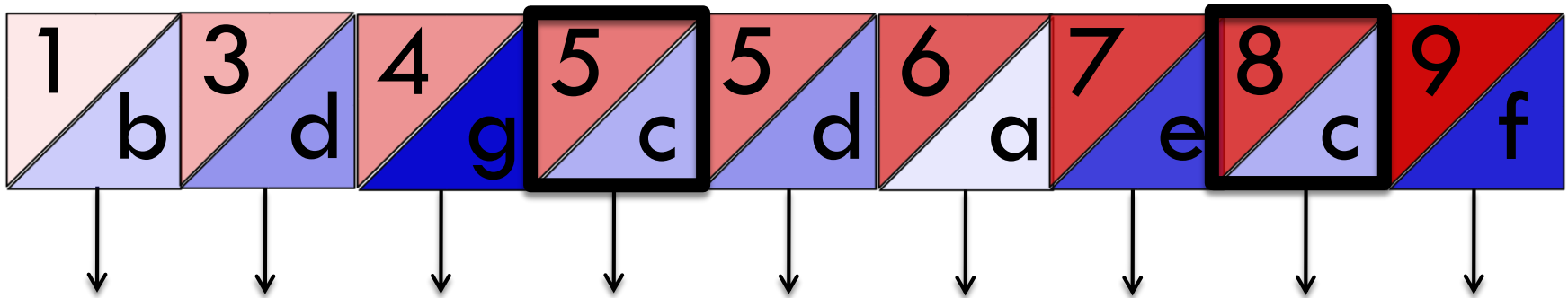
"n" : 3,

Range and Equality

- `db.c.find({x:{$gte:4},y:'c'})`
- `Index {x:1,y:1}`

Range and Equality

? \geq  and  ?

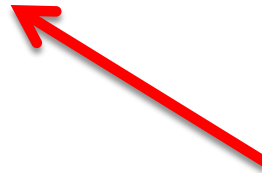


Range and Equality

"nscanned" : 7,

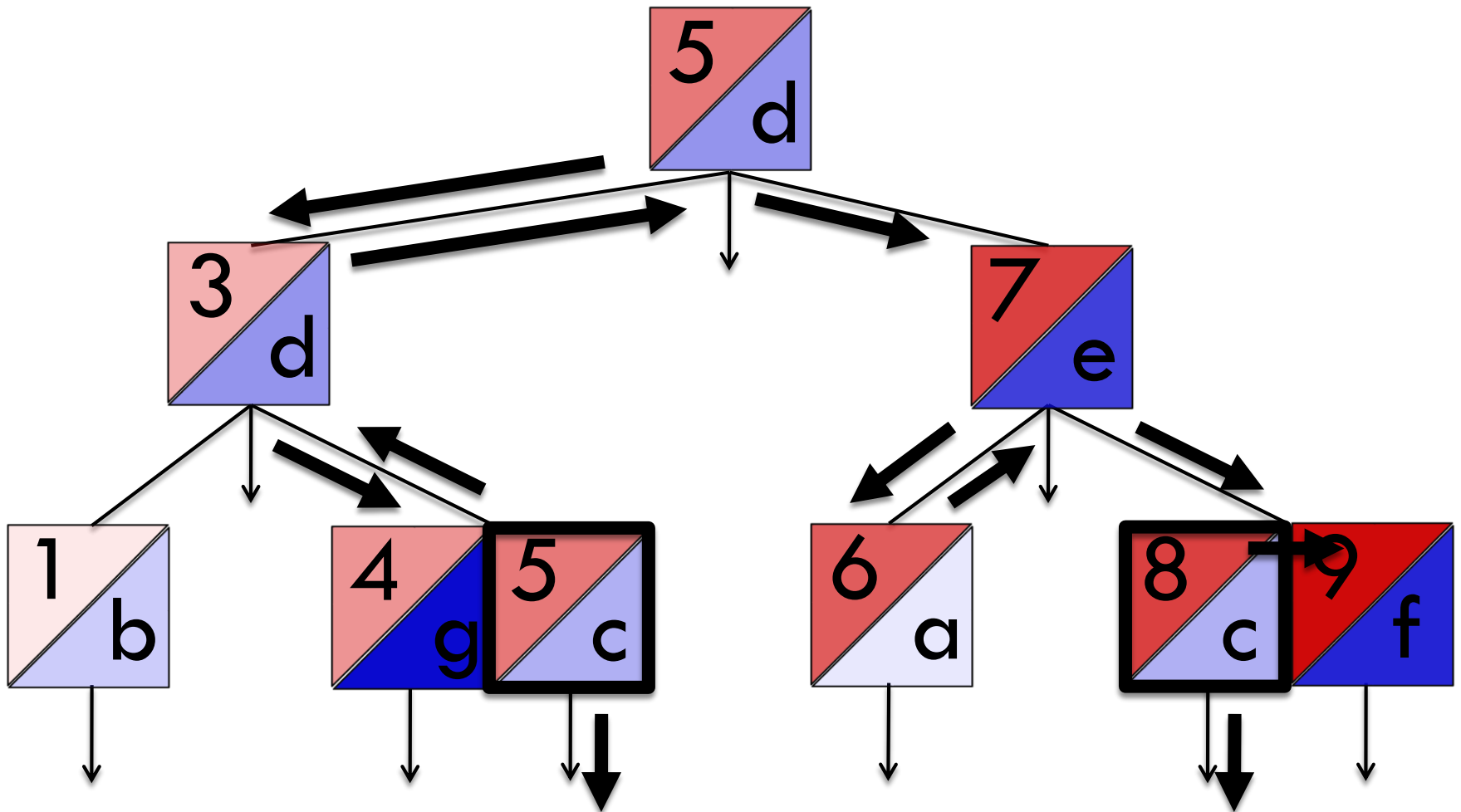
"nscannedObjects" : 2,

"n" : 2,

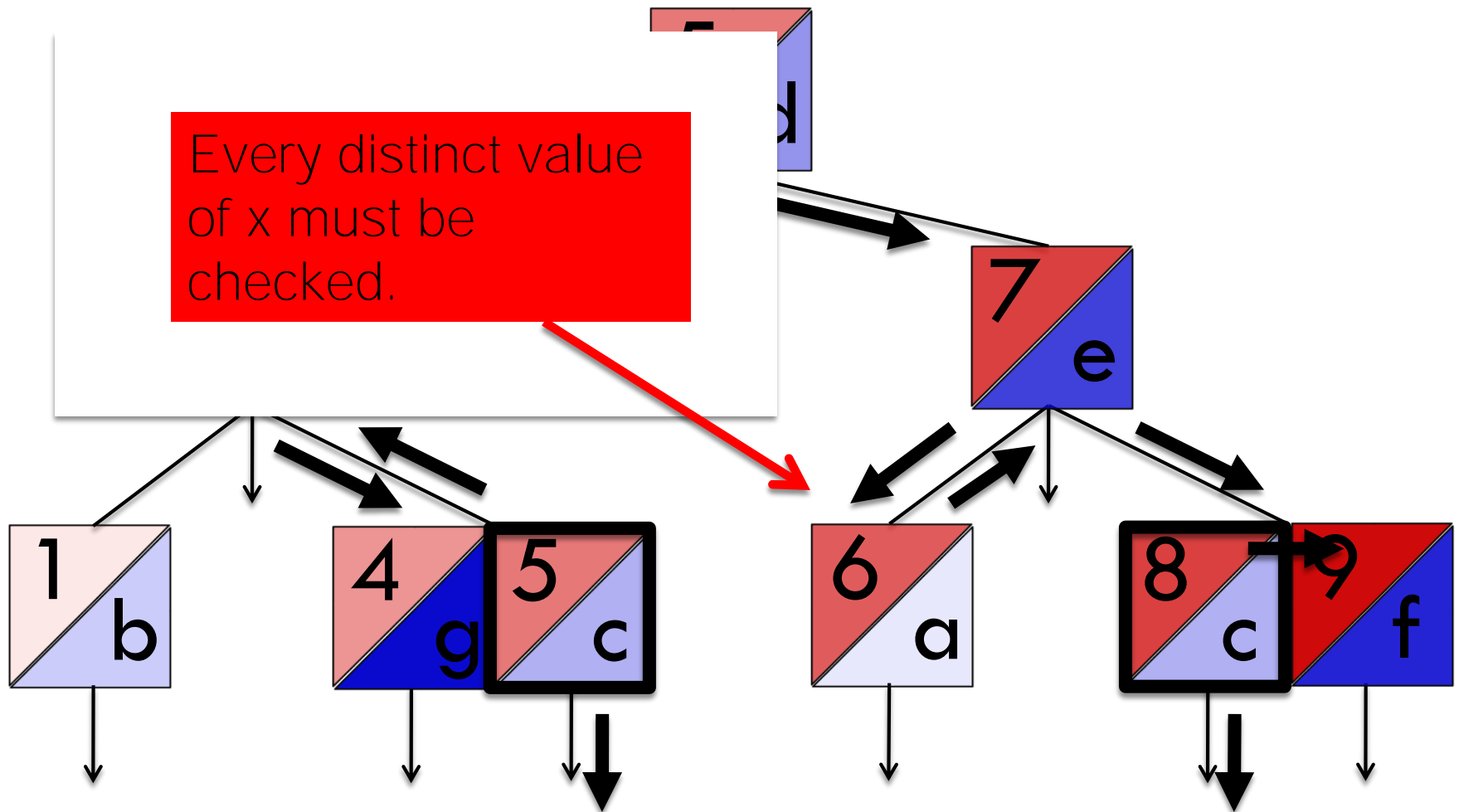


High nscanned
because every
distinct value of x
must be checked.

Range and Equality



Range and Equality

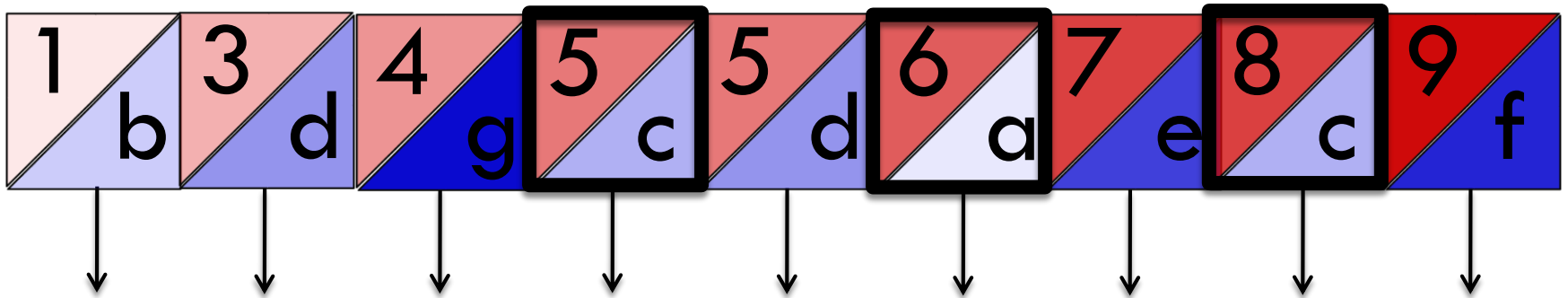


Range and Set

- `db.c.find({x:{$gte:4},y:{$in:['c','a']}})`
- Index `{x:1,y:1}`

Range and Set

? \geq  and  , 



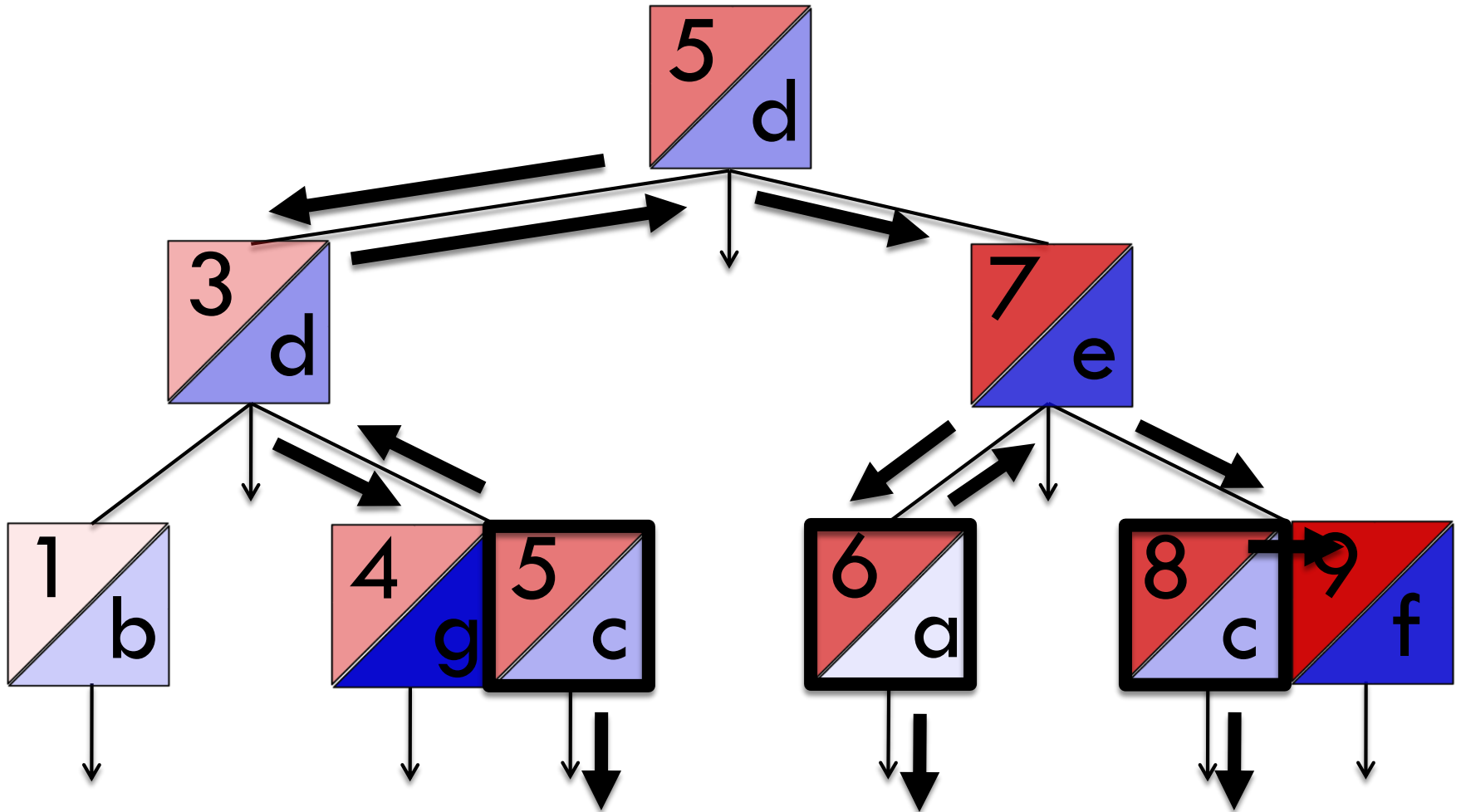
Range and Set

"nscanned" : 7,

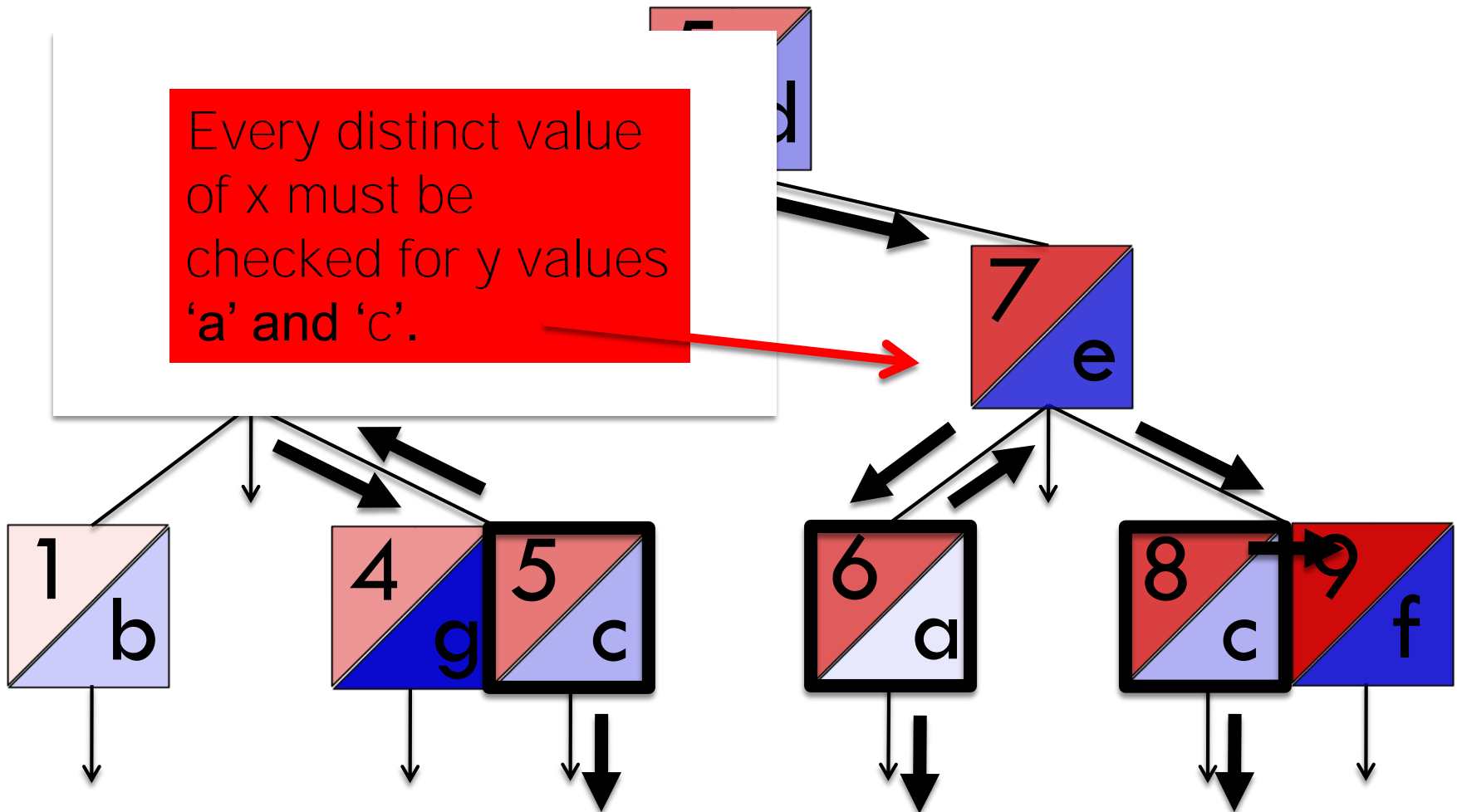
"nscannedObjects" : 3,

"n" : 3,

Range and Set



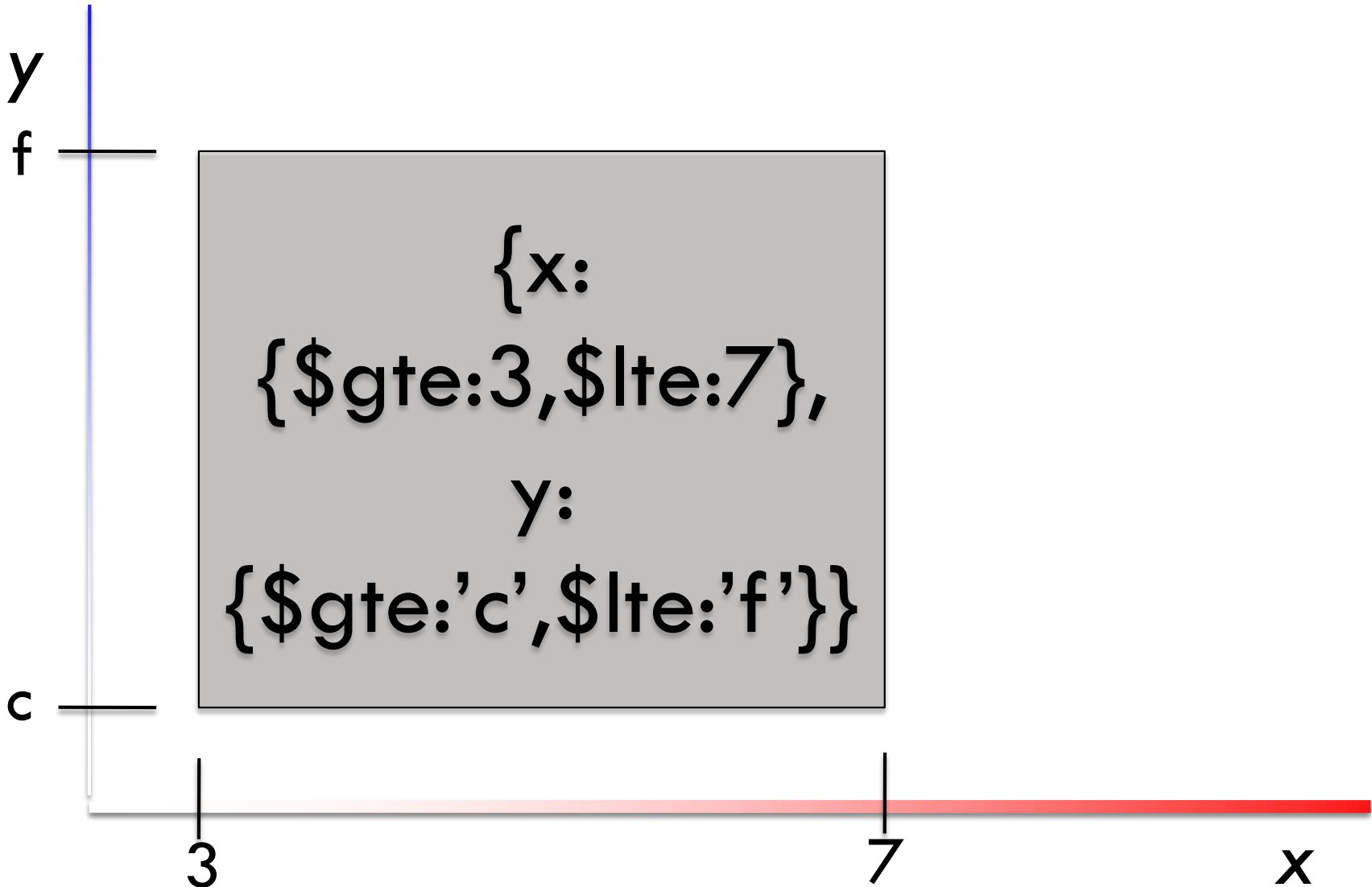
Range and Set



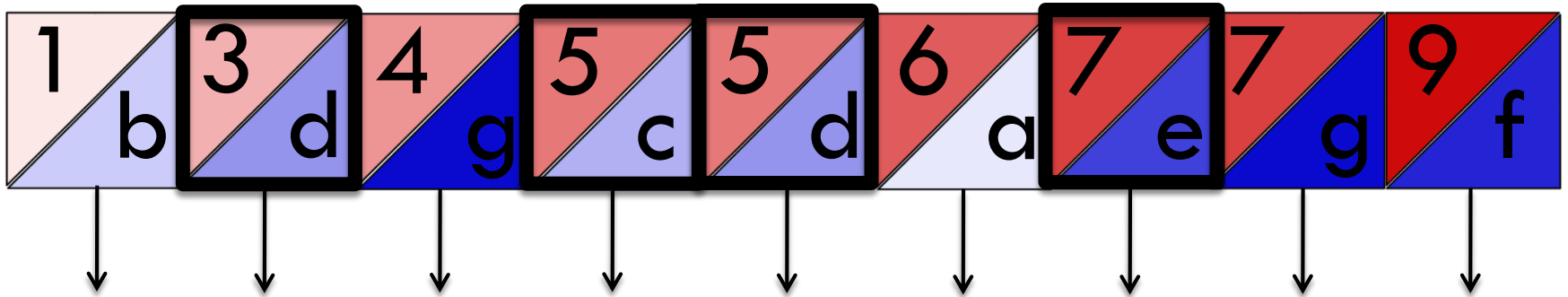
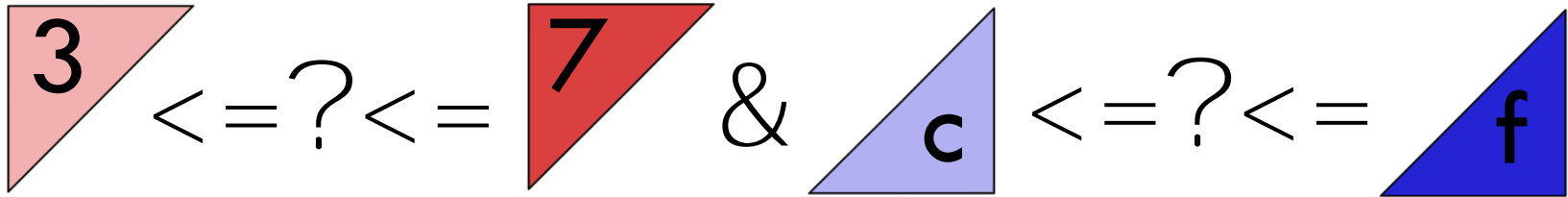
Two Ranges (2D Box)

- `db.c.find({x:{$gte:3,$lte:7},y:{$gte:'c',$lte:'f'}})`
- `Index {x:1,y:1}`

Two Ranges (2D Box)



Two Ranges (2D Box)



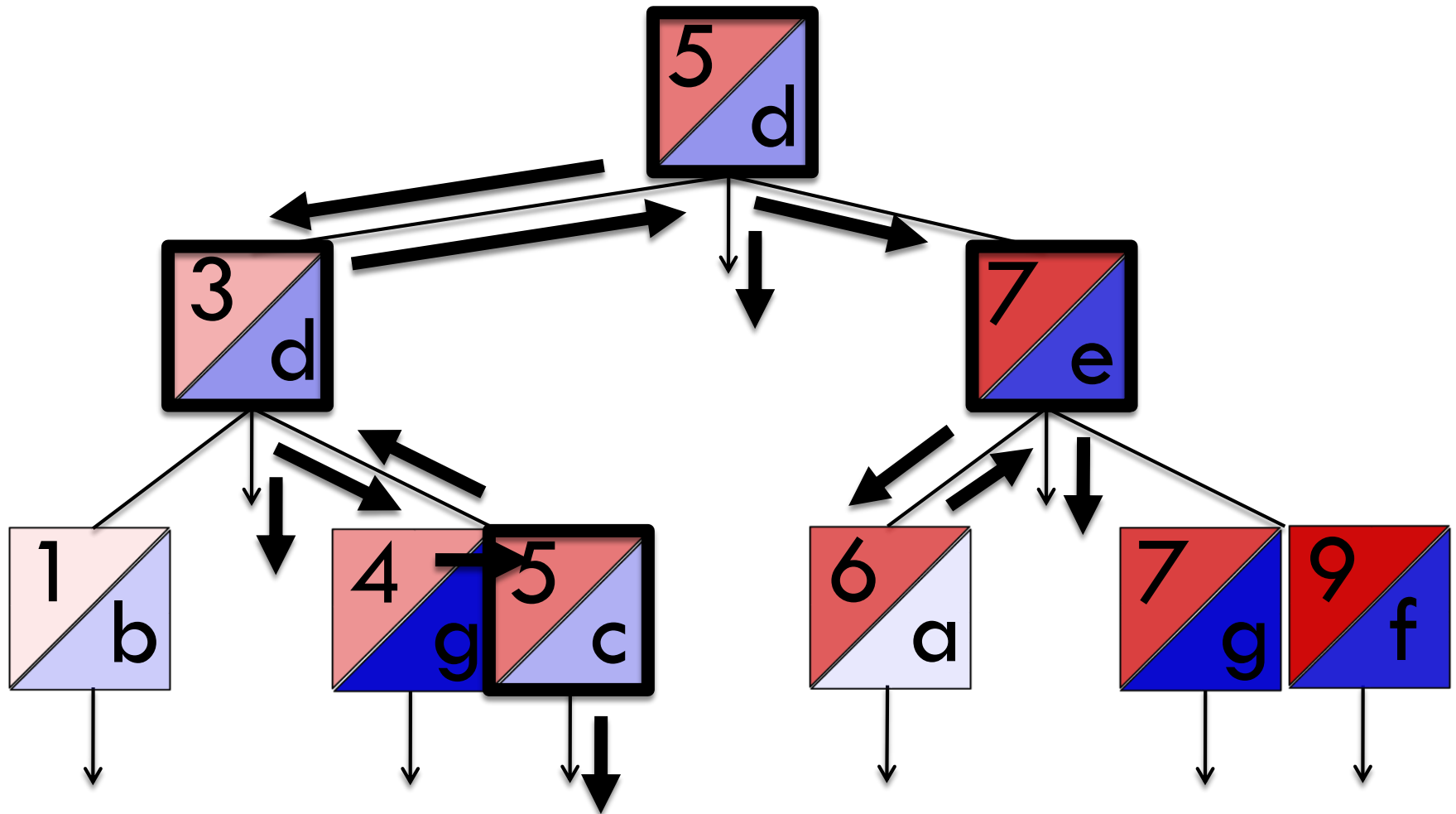
Two Ranges (2D Box)

"nscanned" : 6,

"nscannedObjects" : 4,

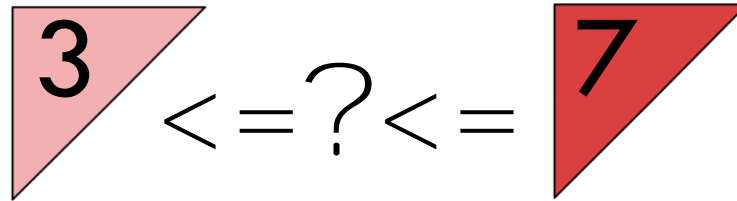
"n" : 4,

Two Ranges (2D Box)

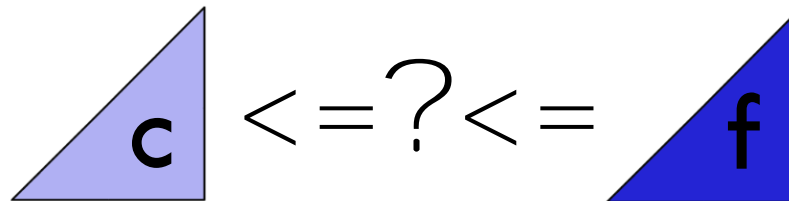


Two Ranges (2D Box)

For every distinct value of x in this range



Scan for every value of y in this range

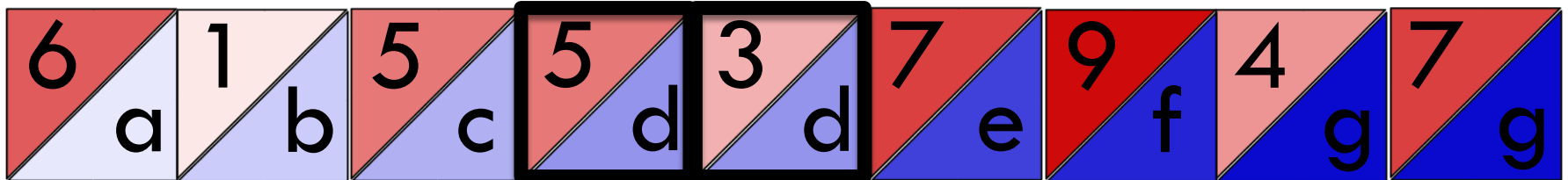
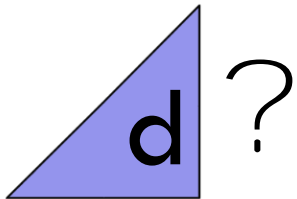
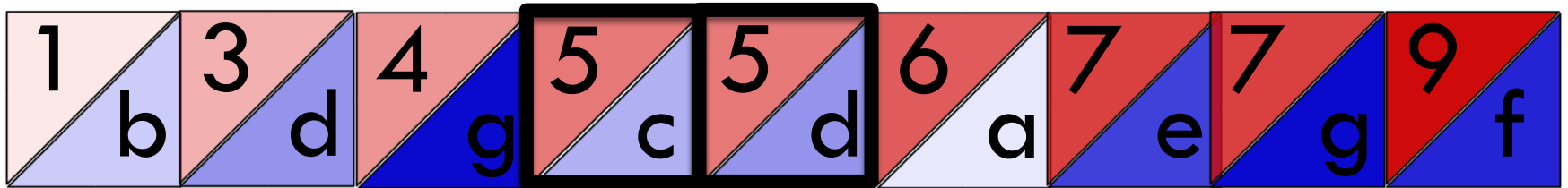
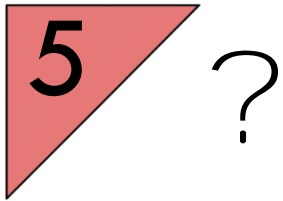


\$or

Disjoint \$or Criteria

- ❑ `db.c.find({$or:[{x:5},{y:'d'}]})`
- ❑ Indexes `{x:1}`, `{y:1}`

Disjoint \$or Criteria



Disjoint \$or Criteria

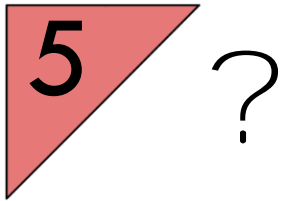
"nscanned" : 4,

"nscannedObjects" : 4,

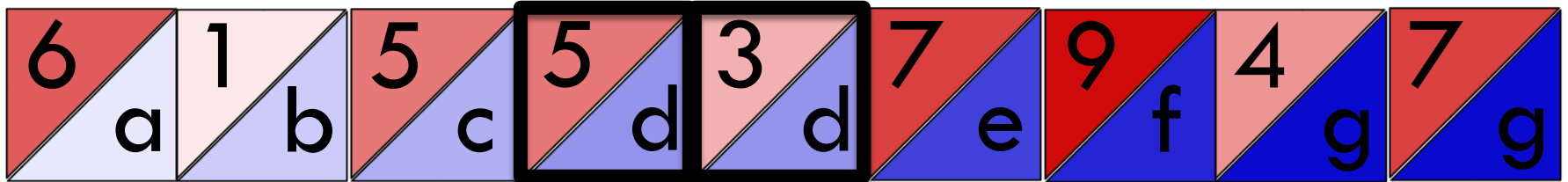
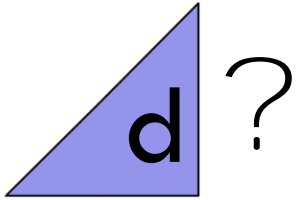
"n" : 3,

"millis" : 1

Disjoint \$or Criteria



Disjoint \$or Criteria



We have already scanned the x index for x:5. So this document was returned already. We don't return it again.



Unindexed \$or Clause

- `db.c.find({$or:[{x:5},{y:'d'}]})`
- Index `{x:1}` (no index on y)

Eliminated \$or Clause

- ❑ `db.c.find({$or:[{x:{$gt:2,$lt:6}},{x:5}]})`
- ❑ Index `{x:1}`

Eliminated \$or Clause

$$\boxed{2} < ? < \boxed{6}$$



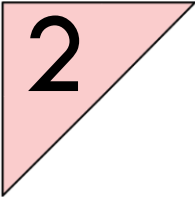
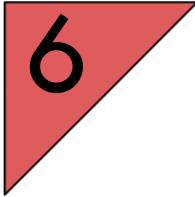
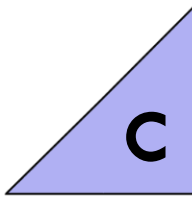
$$\boxed{5} ?$$



Eliminated \$or Clause with Differing Unindexed Criteria

- ❑ `db.c.find({ $or:[{x:{$gt:2,$lt:6},y:'c'},{x:5,y:'d'}]})`
- ❑ Index `{x:1}`

Eliminated \$or Clause with Differing Unindexed Criteria

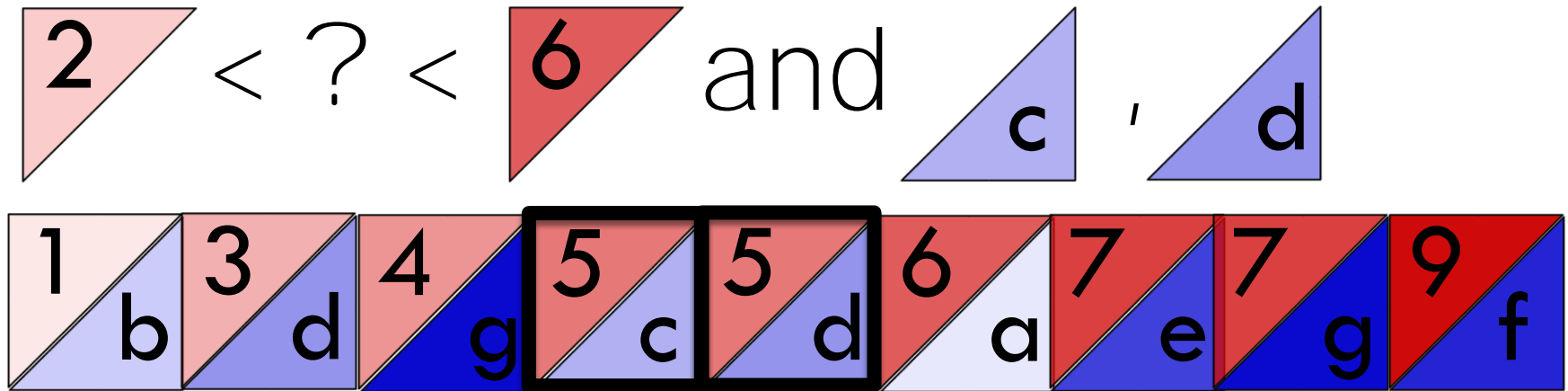
 $< ? <$  and 

1 b	3 d	4 g	5 c	5 d	6 a	7 e	7 g	9 f
--------	--------	--------	--------	--------	--------	--------	--------	--------

 and 

1 b	3 d	4 g	5 c	5 d	6 a	7 e	7 g	9 f
--------	--------	--------	--------	--------	--------	--------	--------	--------

Eliminated \$or Clause with Differing Unindexed Criteria



The index range for the first clause contains the index range for the second clause, so all matching is done using the index range for the first clause.

Overlapping \$or Clauses

- ❑ `db.c.find({$or:[{x:{$gt:2,$lt:6}},{x:{$gt:4,$lt:7}}]})`
- ❑ Index `{x:1,y:1}`

Overlapping \$or\$ Clauses

$$\boxed{2} < ? < \boxed{6}$$



$$\boxed{4} < ? < \boxed{7}$$



Overlapping \$or\$ Clauses

$$\boxed{2} < ? < \boxed{6}$$



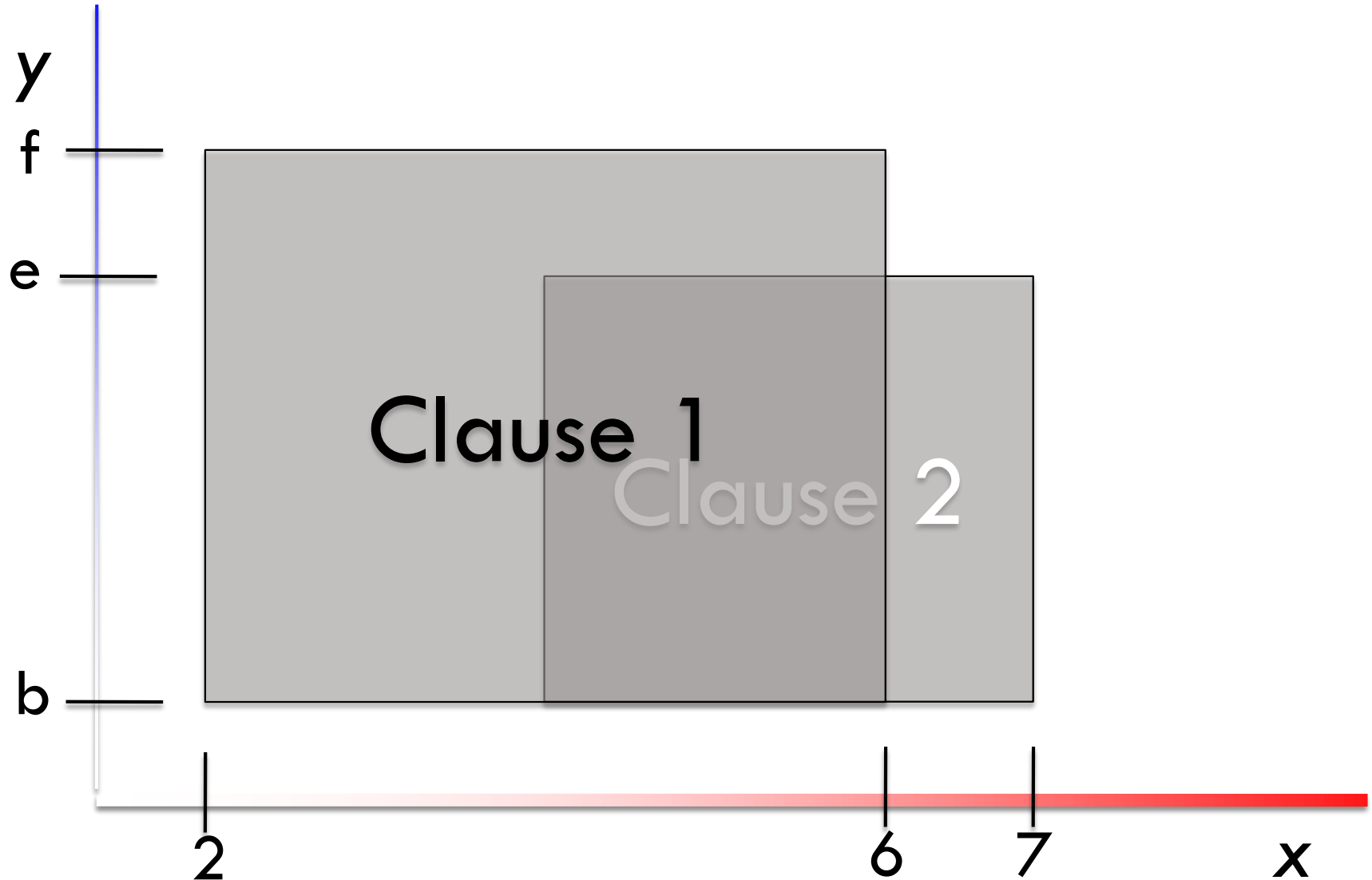
$$\boxed{6} \leq ? < \boxed{7}$$



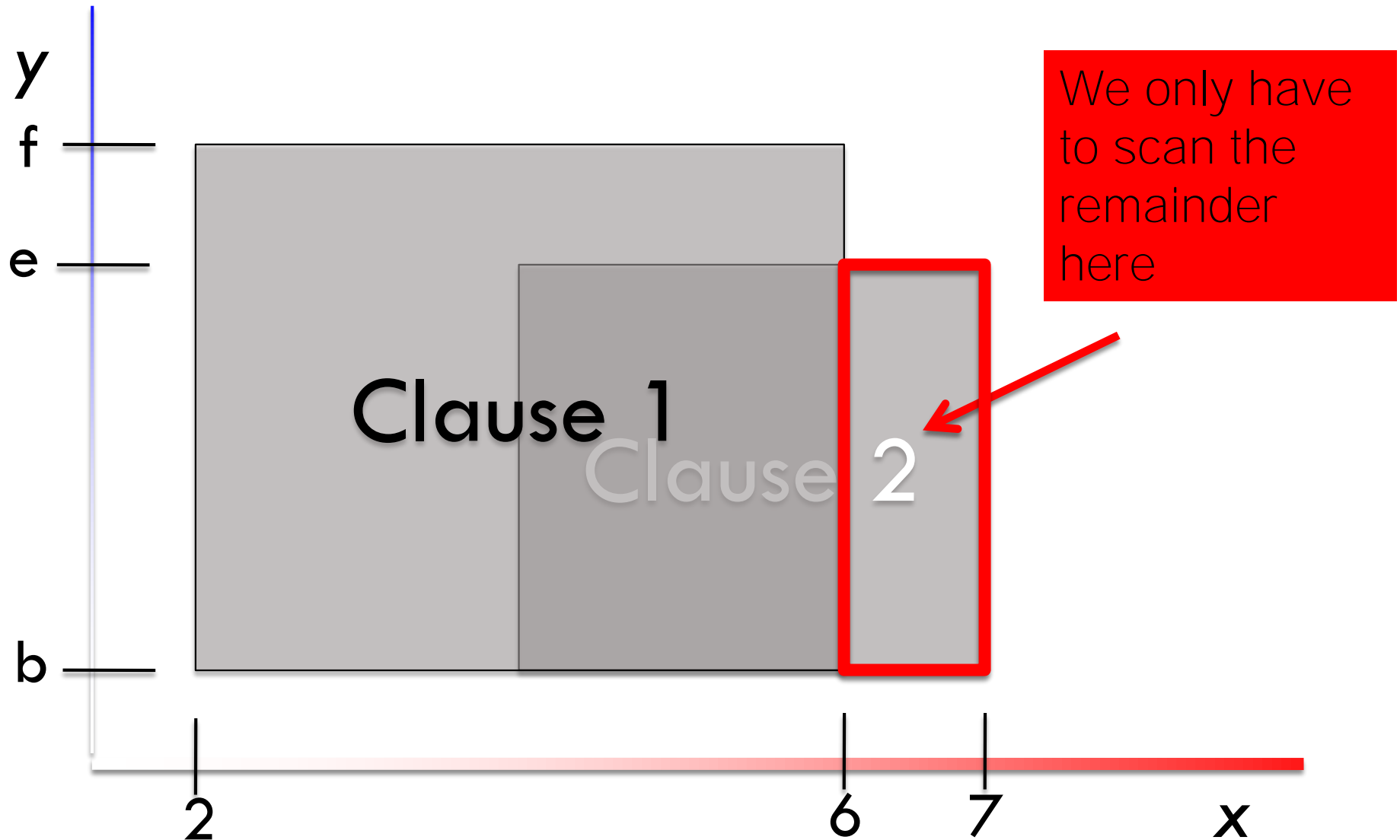
2D Overlapping \$or Clauses

- `db.c.find(`
 `{ $or: [{ x: { $gt: 2, $lt: 6 }, y: { $gt: 'b', $lt: 'f' } }, { x: { $gt: 4, $lt: 7 }, y: { $gt: 'b', $lt: 'e' } }] }`
 `)`
- Index `{x:1,y:1}`

2D Overlapping \$or\$ Clauses

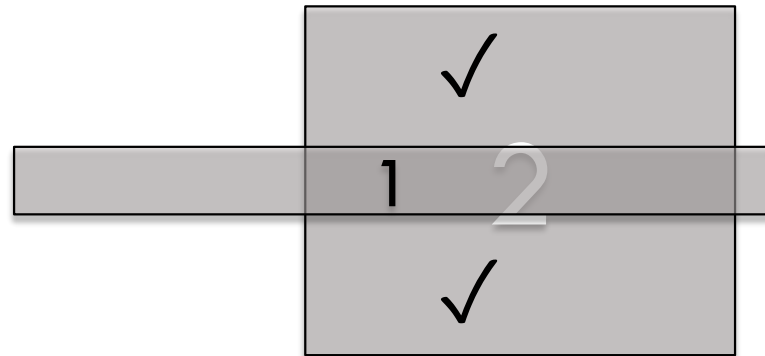
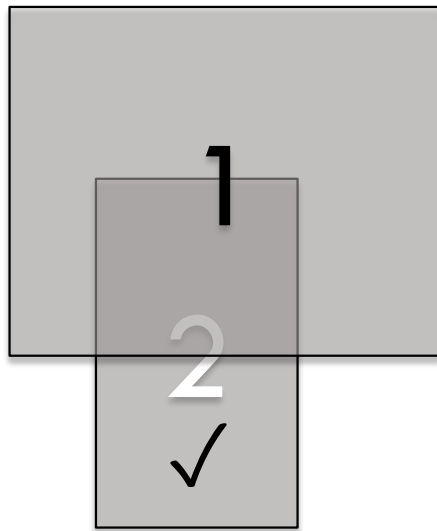


2D Overlapping \$or\$ Clauses



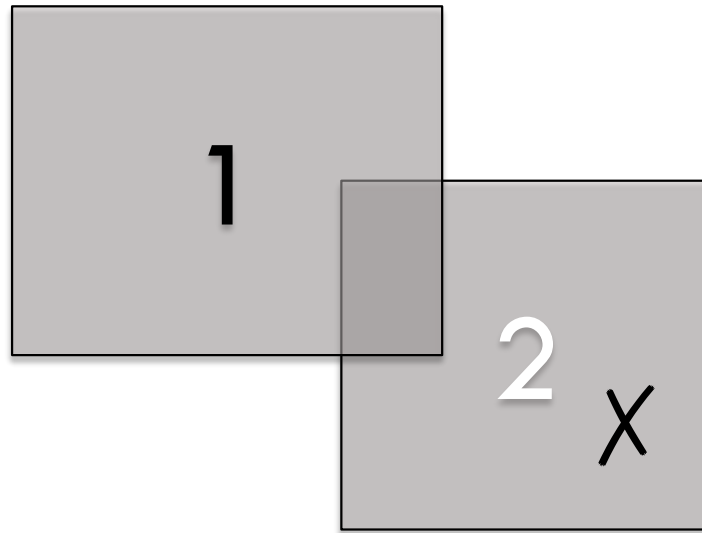
Overlapping \$or\$ Clauses

- Rule of thumb for n dimensions: We subtract earlier clause boxes from current box when the result is a/some box(es).



Overlapping \$or\$ Clauses

- Rule of thumb for n dimensions: We subtract earlier clause boxes from current box when the result is a/some box(es).



\$or TODO

- Use indexes on \$or fields to satisfy a sort specification SERVER-1205
- Use full query optimizer to select \$or clause indexes in getMore SERVER-1215
- Improve index range elimination (handling some cases where remainder is not a box)

Automatic Index Selection (Query Optimizer)

Optimal Index

- `find({x:5})`
 - ▣ `Index {x:1}`
 - ▣ `Index {x:1,y:1}`
- `find({x:5}).sort({y:1})`
 - ▣ `Index {x:1,y:1}`
- `find({}).sort({x:1})`
 - ▣ `Index {x:1}`
- `find({x:{$gt:1,$lt:7}}).sort({x:1})`
 - ▣ `Index {x:1}`

Optimal Index

- Rule of Thumb
 - ▣ No scanAndOrder
 - ▣ All fields with index useful constraints are indexed
 - ▣ If there is a range or sort it is the last field of the index used to resolve the query
- If multiple optimal indexes exist, one chosen arbitrarily.

Optimal Index

- These same criteria are useful when you are designing your indexes.

Multiple Candidate Indexes

- `find({x:4,y:'a'})`
 - ▣ Index `{x:1}` or `{y:1}`?
- `find({x:4}).sort({y:1})`
 - ▣ Index `{x:1}` or `{y:1}`?
 - ▣ Note: `{x:1,y:1}` is optimal
- `find({x:{$gt:2,$lt:7},y:{$gt:'a',$lt:'f'}})`
 - ▣ Index `{x:1,y:1}` or `{y:1,x:1}`?

Multiple Candidate Indexes

- The only index selection criterion is nscanned
- find({x:4,y:'a'})
 - ▣ Index {x:1} or {y:1} ?
 - ▣ If fewer documents match {y:'a'} than {x:4} then nscanned for {y:1} will be less so we pick {y:1}
- find({x:{>2,<7},y:{>'b',<'f'}})
 - ▣ Index {x:1,y:1} or {y:1,x:1} ?
 - ▣ If fewer distinct values of $2 < x < 7$ than distinct values of $'b' < y < 'f'$ then {x:1,y:1} chosen (rule of thumb)

Multiple Candidate Indexes

- The only index selection criterion is nscanned
- Pretty good, but doesn't cover every case, eg
 - ▣ Cost of scanAndOrder vs ordered index
 - ▣ Cost of loading full document vs just index key
 - ▣ Cost of scanning adjacent btree keys vs non adjacent keys/documents

Competing Indexes

- At most one query plan per index
- Run in interleaved fashion
- Plans kept in a priority queue ordered by nscanned.
We always continue progress on plan with lowest nscanned.

Competing Indexes

- Run until one plan returns all results or enough results to satisfy the initial query request (based on soft limit spec / data size requirement for initial query).
- We only allow plans to compete in initial query. In `getMore`, we continue reading from the index cursor established by the initial query.

“Learning” a Query Plan

- When an index is chosen for a query the query’s “pattern” and nscanned are recorded
 - ▣ find({x:3,y:'c'})
 - {Pattern: {x:'equality', y:'equality'}, Index: {x:1}, nscanned: 50}
 - ▣ find({x:{\$gt:5},y:{\$lt:'z'}})
 - {Pattern: {x:'gt bound', y:'lt bound'}, Index: {y:1}, nscanned: 500}

“Learning” a Query Plan

- When a new query matches the same pattern, the same query plan is used
 - ▣ `find({x:5,y:'z'})`
 - Use index {x:1}
 - ▣ `find({x:{$gt:20},y:{$lt:'b'}})`
 - Use index {y:1}

“Un-Learning” a Query Plan

- 100 writes to the collection
- Indexes added / removed

Bad Plan Insurance

- If nscanned for a new query using a recorded plan is much worse than the recorded nscanned for an earlier query with the same pattern, we start interleaving other plans with the current plan.
- Currently “much worse” means 10x

Query Planner

- Ad hoc heuristics in some cases
- Seem to work decently in practice