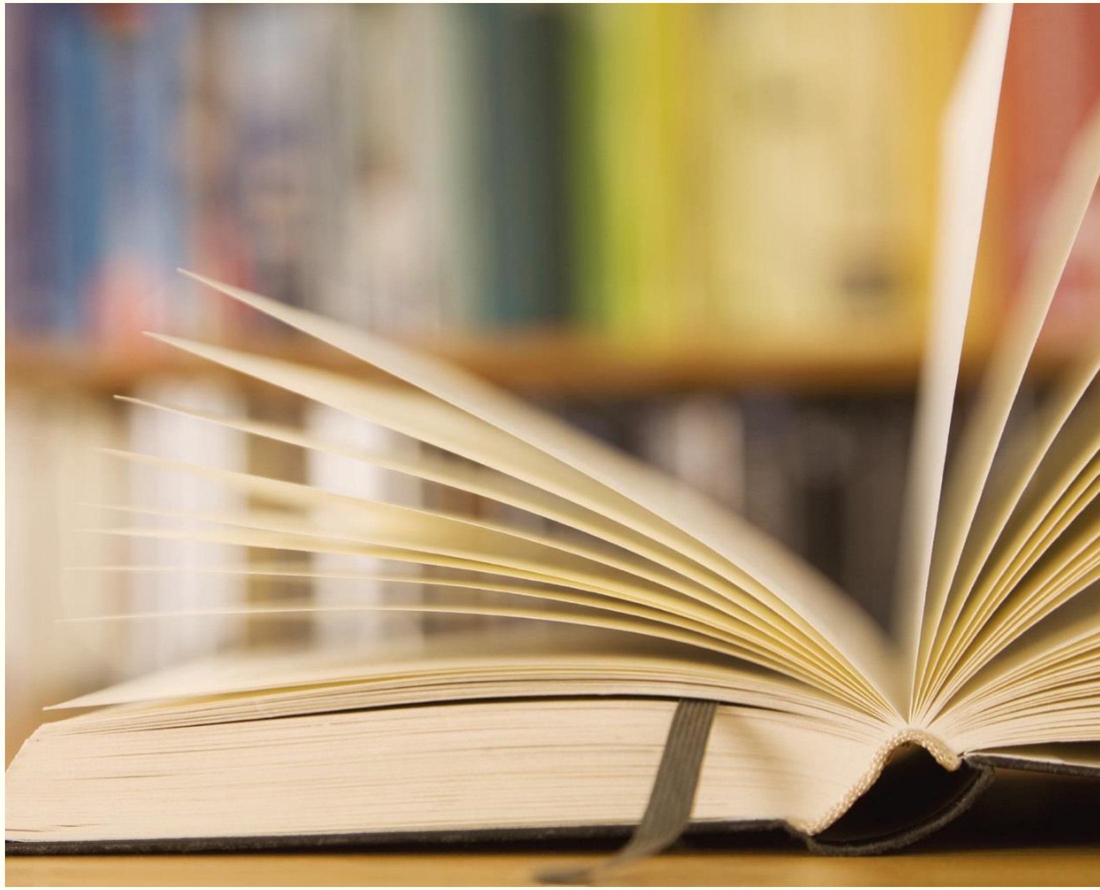
NoSQL

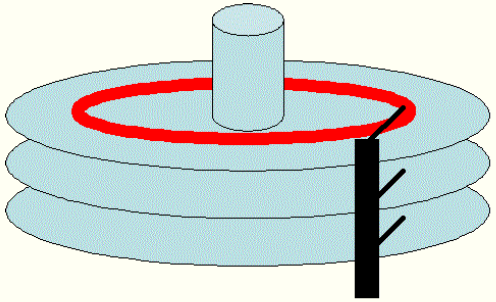
Column Store, Document DB



Row Store and Column Store

- Most of the queries does not process all the attributes of a particular relation.

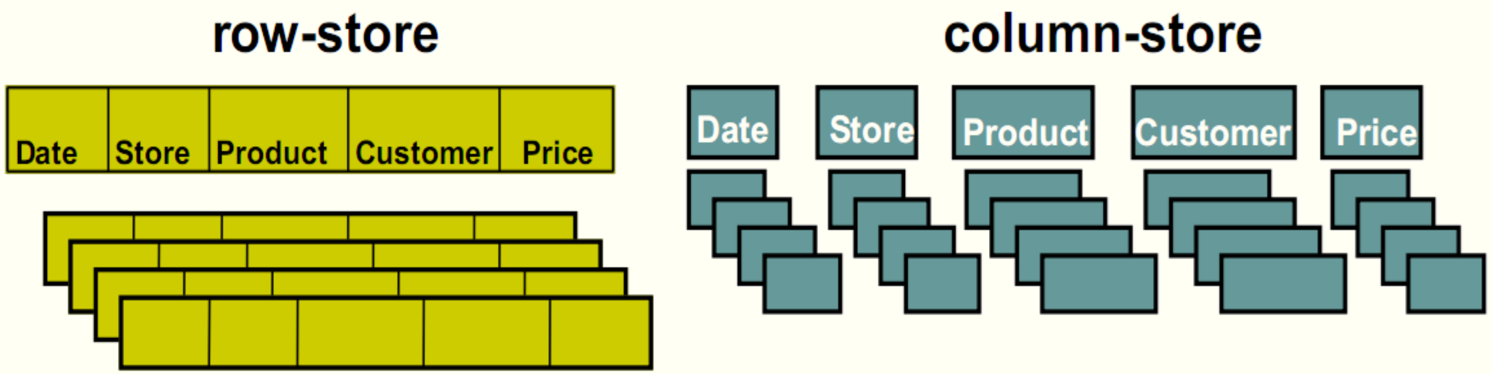
- For example the query
  - ✓ Select c.name and c.address
  - ✓ From CUSTOMER as c
  - ✓ Where c.region=Pullman;



- Only process three attributes of the relation CUSTOMER. But the customer relation can have more than three attributes.

- more I/O efficient for read-only queries as they read, only those attributes which are accessed by a query.

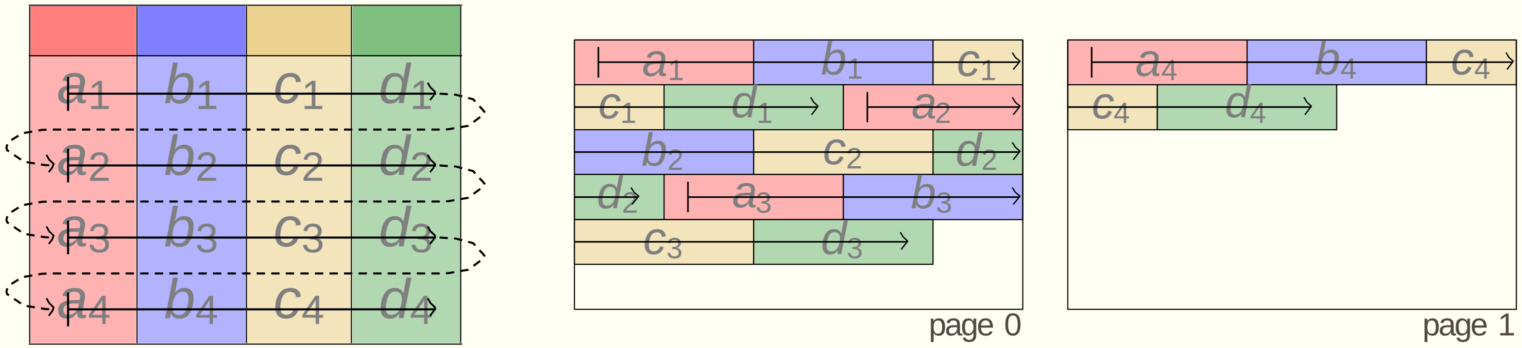
Row Store and Column Store



- In row store data are stored in the disk tuple by tuple.
- Where in column store data are stored in the disk column by column

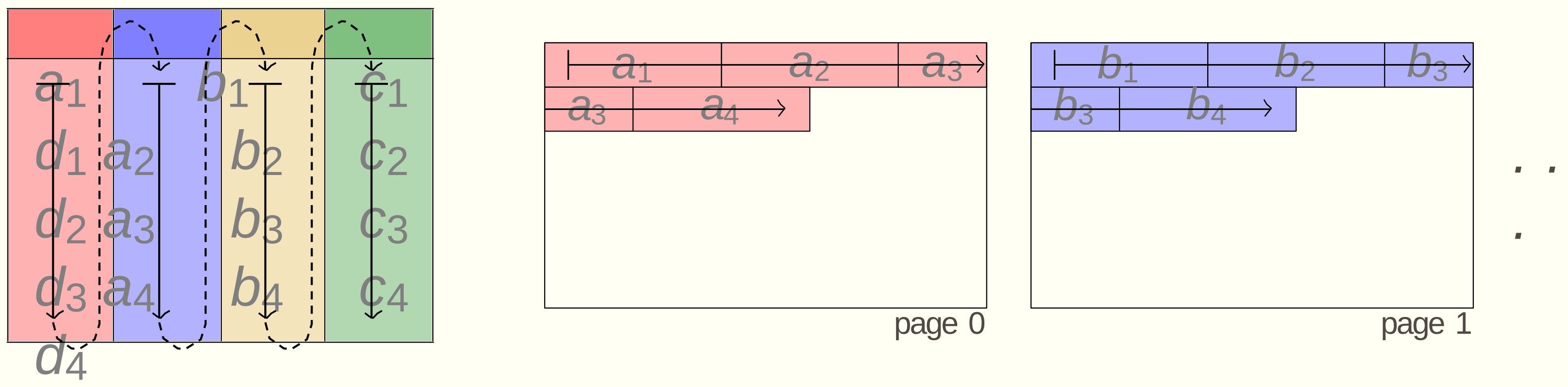
Row Stores

- In a row-store, a.k.a. row-wise storage or n-ary storage model, NSM:
  all rows of a table are stored sequentially on a database page.



Column-stores

- a.k.a. column-wise storage or decomposition storage model, DSM:



The Effect on Query Processing

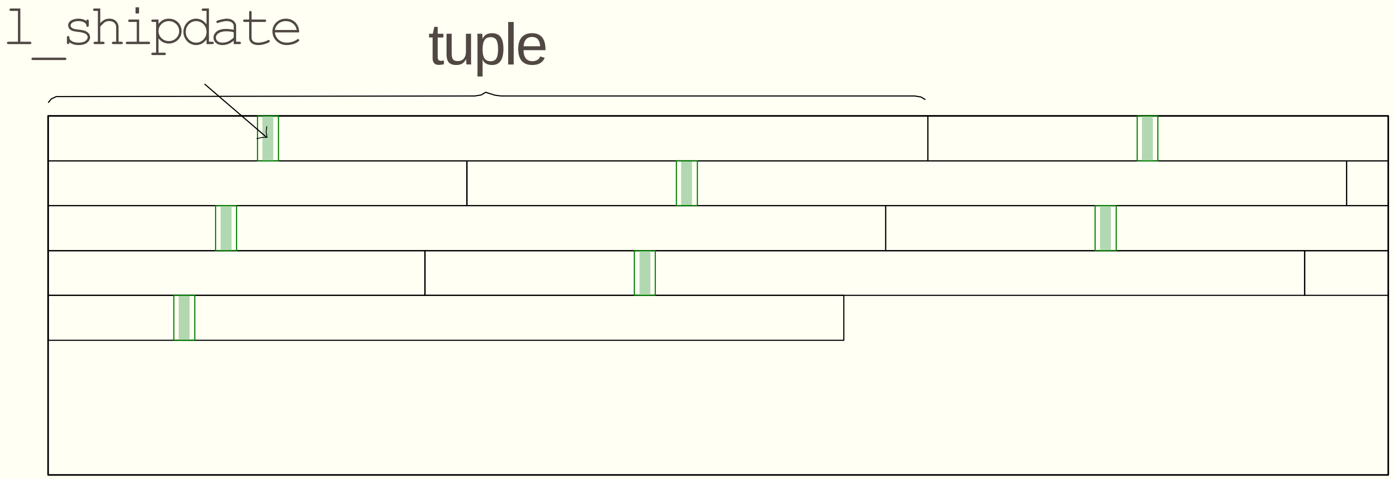
Consider, e.g., a selection query:

```
SELECT COUNT(*)
FROM lineitem
WHERE l_shipdate = "2017-10-19"
```

This query typically involves a full table scan.

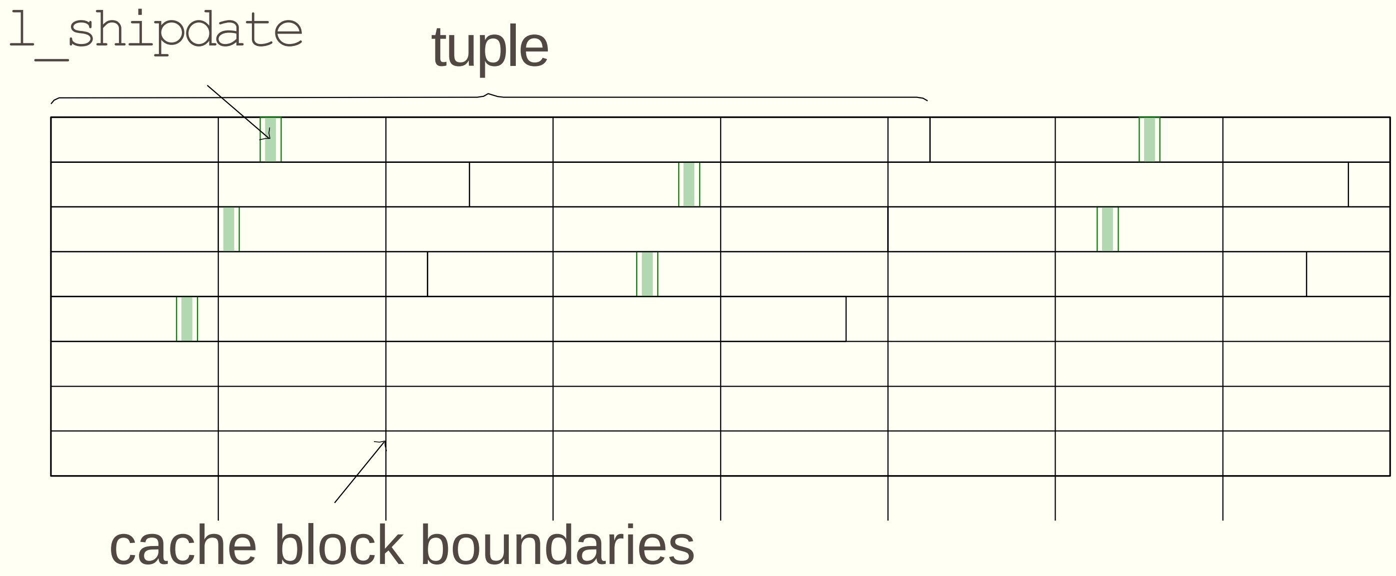
A Full Table Scan in a Row Store

- In a row-store, all rows of a table are stored sequentially on a database page.



A full table scan in a row-store

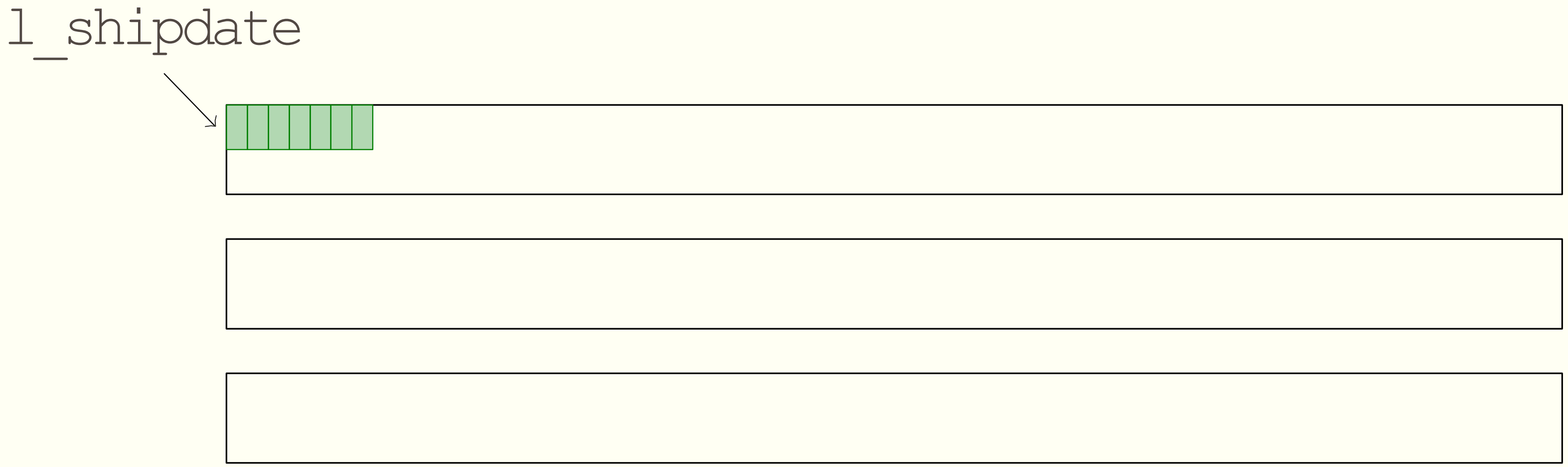
- In a row-store, all rows of a table are stored sequentially on a database page.



With every access to a l_shipdate field, load a large amount of irrelevant information into the cache.

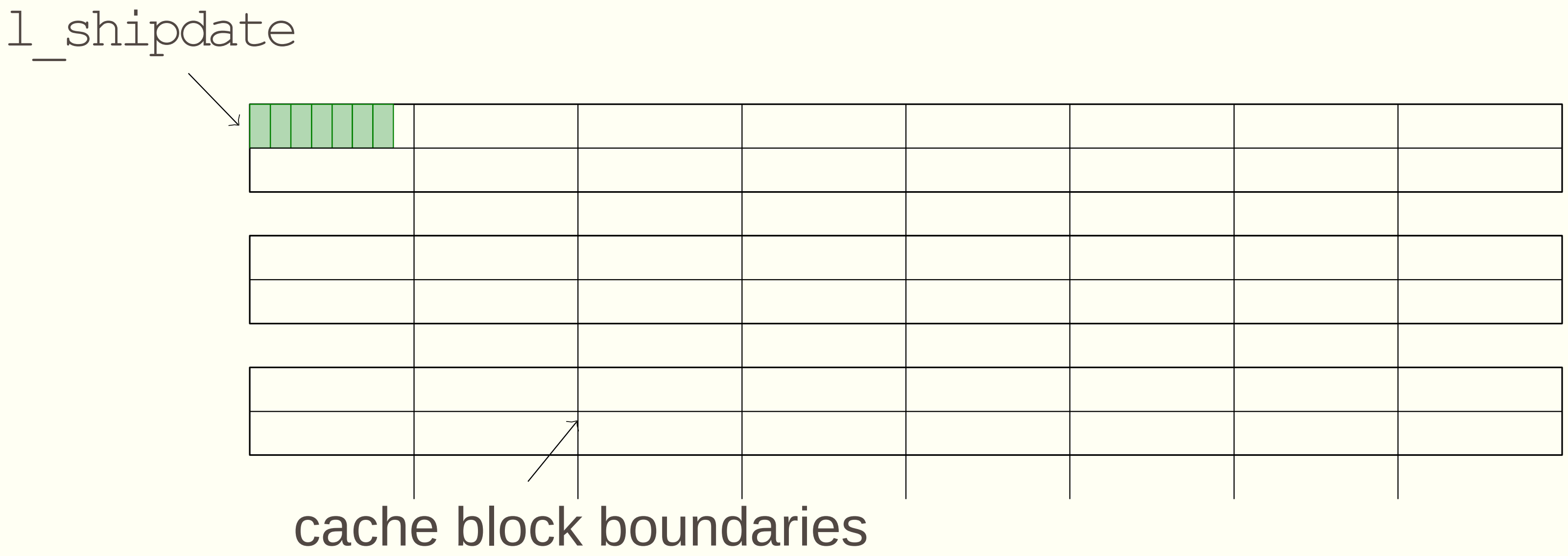
A “Full Table Scan” on a Column Store

- In a column-store, all values of one column are stored sequentially on a database page.



A “Full Table Scan” on a Column Store

- In a column-store, all values of one column are stored sequentially on a database page.



All data loaded into caches by a “`l_shipdate` scan” is now actually **relevant** for the query.

Column-store Advantages

- Less data has to be fetched from memory.
- Amortize cost for fetch over more tuples.
  - The same arguments hold also for in-memory based systems (we will see soon).
  - Additional benefit: Data compression might work better.

Why Column Store

- Can be significantly faster than row stores for some applications
  - Fetch only required columns for a query
  - Better cache effects
  - Better compression (similar attribute values within a column)
- But can be slower for other applications
  - OLTP with many row inserts, ..
- Long war between the column store and row store camps :-)

Row Store and Column Store

Row Store	Column Store
(+) Easy to add/modify a record	(+) Only need to read in relevant data
(-) Might read in unnecessary data	(-) Tuple writes require multiple accesses

Column stores are suitable for read-mostly, read-intensive, large data repositories

Column Stores - Data Model

- Standard relational logical data model
  - EMP(name, age, salary, dept)
  - DEPT(dname, floor)
- Table – collection of projections
- Projection – set of columns
- Horizontally partitioned into segments with segment identifier

EMP1

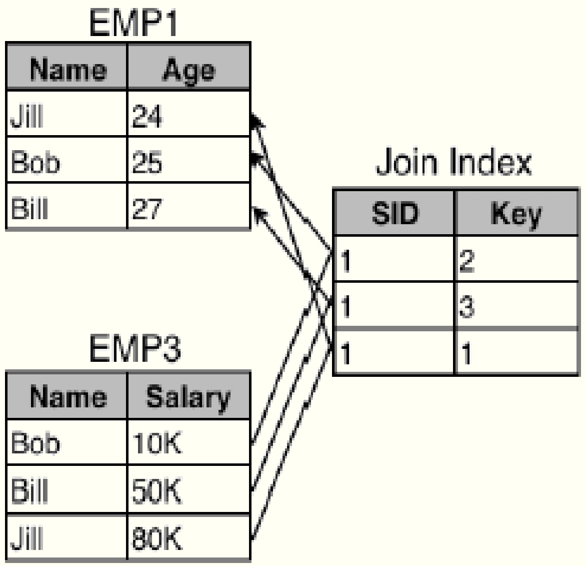
Name	Age
Jill	24
Bob	25
Bill	27

EMP3

Name	Salary
Bob	10K
Bill	50K
Jill	80K

Column Stores - Data Model

- To answer queries, projections are joined using Storage keys and join indexes
- Storage Keys:
  - Within a segment, every data value of every column is associated with a unique Skey
  - Values from different columns with matching Skey belong to the same logical row



Query Execution - Operators

- **Select:** Same as relational algebra, but produces a bit string
- **Project:** Same as relational algebra
- **Join:** Joins projections according to predicates

- **Aggregation:** SQL like aggregates
- **Sort:** Sort all columns of a projection
- **Decompress:** Converts compressed column to uncompressed representation
- **Mask**(Bitstring B, Projection Cs) => emit only those values whose corresponding bits are 1
- **Concat:** Combines one or more projections sorted in the same order into a single projection
- **Permute:** Permutes a projection according to the ordering defined by a join index
- **Bitstring operators:** Band – Bitwise AND, Bor – Bitwise OR, Bnot – complement

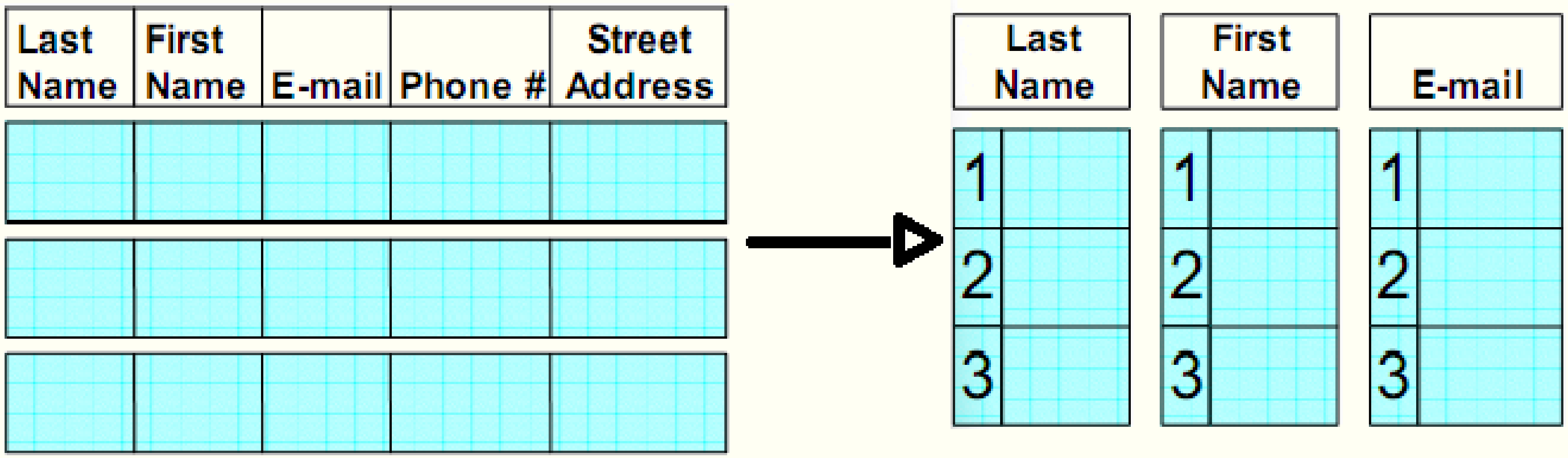
Row Store Vs Column Store

- One can obtain the performance benefits of a column-store using a row-store by making some changes to the physical structure of the row store.
  - Vertically partitioning
  - Using index-only plans
  - Using materialized views

Vertical Partitioning

- Process:
    - Full Vertical partitioning of each relation
        - Each column =1 Physical table
            - by adding integer position column to every table
    - Join on Position for multi column fetch
- Problems:
    - "Position" - Space and disk bandwidth
    - Header for every tuple – further space wastage
        - e.g. 24 byte overhead in PostgreSQL

Vertical Partitioning: Example



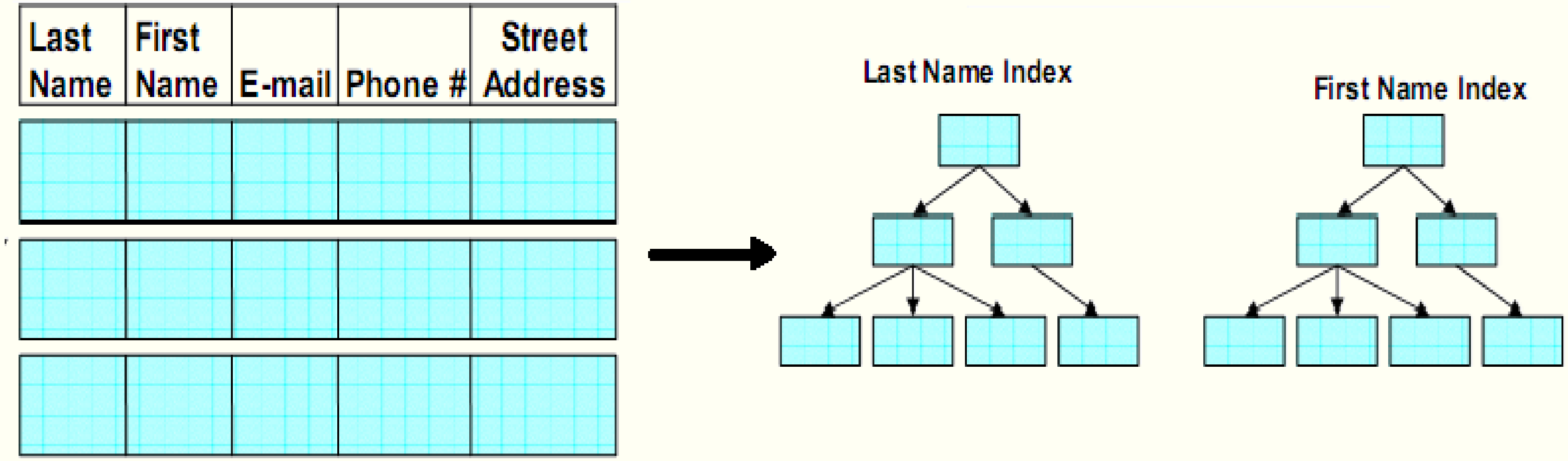
Vertical Partitioning

Index-only plans

- Process:
  - Add B+Tree index for every Table.column
  - Plans never access the actual tuples on disk
  - Headers are not stored, so per tuple overhead is less

- Problem:
  - Separate indices may require full index scan, which is slower
  - Eg: SELECT AVG(salary)
    FROM emp
    WHERE age > 40
  - Composite index with (age, salary) key helps.

Index-Only Plans



Index Every Column

Materialized Views

- Process:
  - Create ‘optimal' set of MVs for given query workload
  - Objective:
    - Provide just the required data
    - Avoid overheads
    - Performs better
- Expected to perform better than other two approach

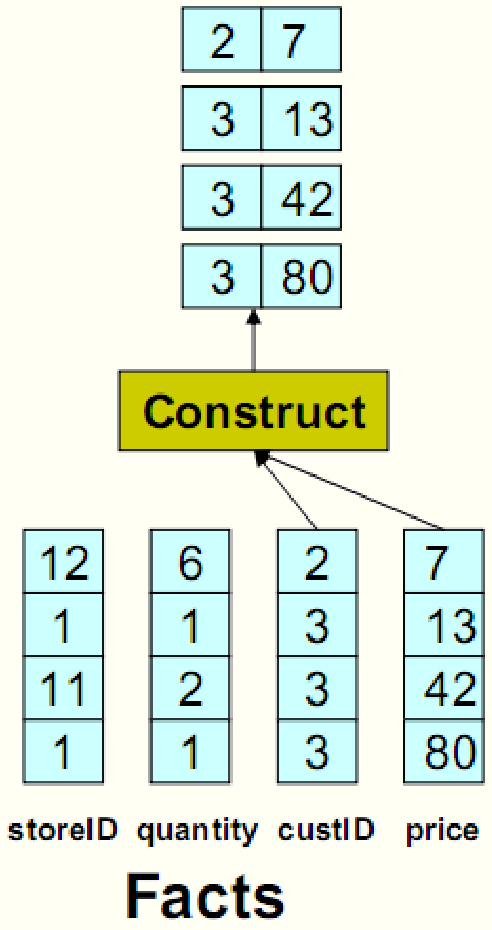
- Problems:
  - Practical only in limited situation
  - Require knowledge of query workloads in advance

Materialized Views: Example

- Select F.custID

  from Facts as F

  where F.price>20

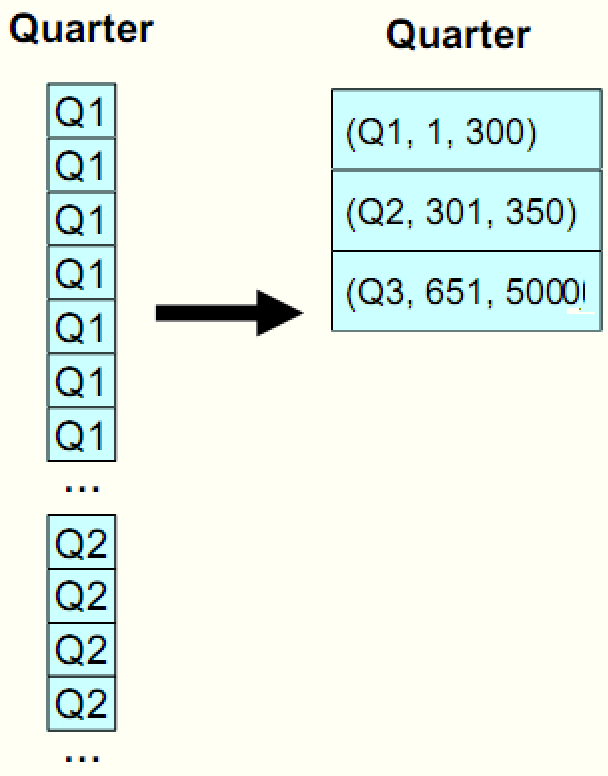


Optimizing Column oriented Execution

- Different optimization for column-oriented database
  - Compression
  - Late Materialization
  - Block Iteration

Compression

- column can be super-compressible
- E.g. Run length encoding



Compression

- Low information entropy (high data value locality) leads to High compression ratio
- Advantage
  - Disk Space is saved
  - Less I/O
  - CPU cost decrease if we can perform operation without decompressing
- Light weight compression schemes do better

Late Materialization

- Most query results entity-at-a-time not column-at-a-time

- Idea: Delay Tuple Construction
- Might avoid constructing it altogether
- Intermediate position lists might need to be constructed
  - SELECT R.a FROM R WHERE R.c = 5 AND R.b = 10

  - Output of each predicate is a bit string
  - Perform Bitwise AND
  - Use final position list to extract R.a

Block Iteration

- Operators operate on blocks of tuples at once
- Iterate over blocks rather than tuples
- If column is fixed width, it can be operated as an array
- Minimizes per-tuple overhead
- Exploits potential for parallelism
- Can be applied even in Row stores – IBM DB2 implements it

Document DB

- Documents are the main concept.
- A Document-oriented database stores and retrieves documents (XML, JSON, BSON and so on).
- Documents are:
  - Self-describing
  - Hierarchical tree data structures (maps, collection and scalar values)

What is a Document DB?

- Document databases store documents in the value part of the key-value store

```
{
  name: "sue",                        ← field: value
  age: 26,                            ← field: value
  status: "A",                        ← field: value
  groups: [ "news", "sports" ]        ← field: value
}
```

- Documents may have different attributes
- Different from relational databases where columns stores the same type of values or null

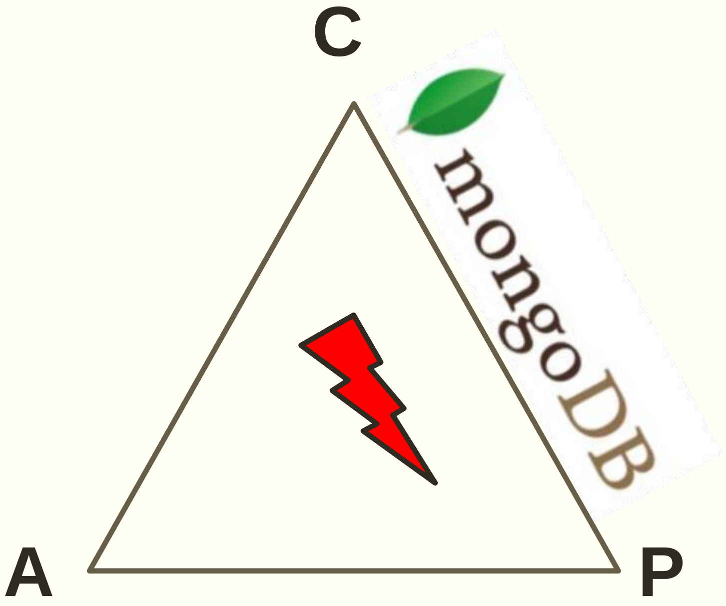
MongoDB

- MongoDB Features
  - Document-Oriented storage
  - Index Support
  - Replication & Availability
  - Auto-Sharding
  - Ad-hoc Querying
  - Fast In-Place Updates
  - Map/Reduce functionality

MongoDB: CAP approach

- **Focus on Consistency and Partition tolerance**

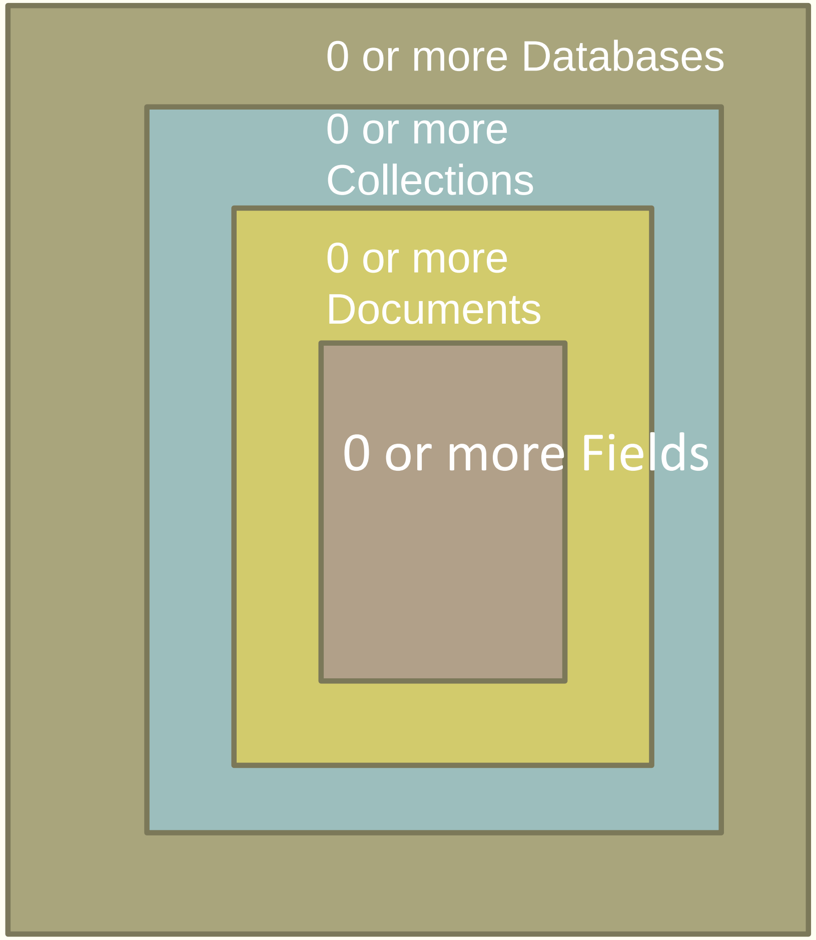
- **C**onsistency
    - all replicas contain the same version of the data
- **A**vailability
    - system remains operational on failing nodes
- **P**artition tolarence
    - multiple entry points
    - system remains operational on system split



CAP Theorem:
satisfying all three at the same time is impossible

MongoDB: Hierarchical Objects

- A MongoDB instance may have zero or more ‘databases’
- A database may have zero or more ‘collections’.
- A collection may have zero or more ‘documents’.
- A document may have one or more ‘fields’.
- MongoDB ‘Indexes’ function much like their RDBMS counterparts.

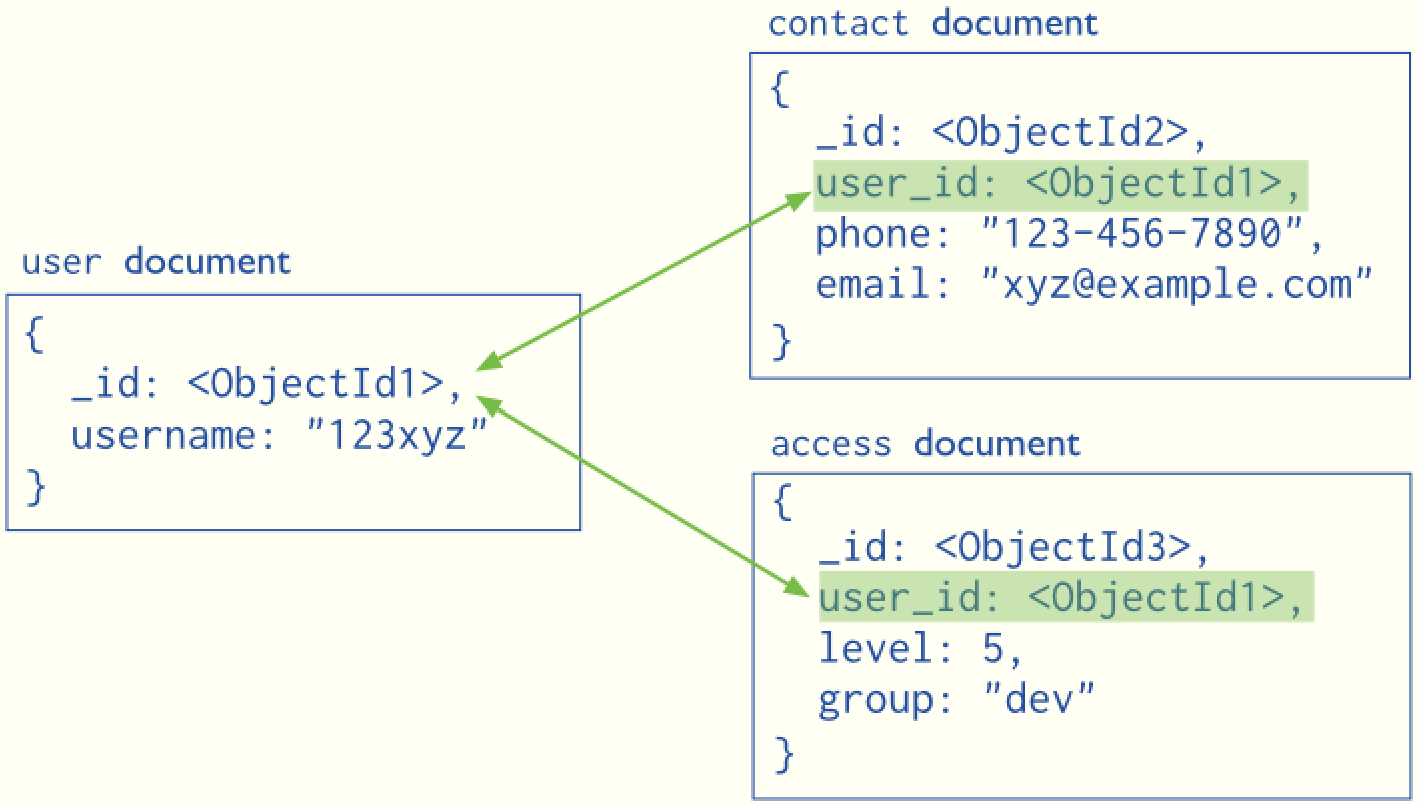


The _id Field

- By default, each document contains an _id field. This field has a number of special characteristics:
    - Value serves as primary key for collection.
    - Value is unique, immutable, and may be any non-array type.
    - Default data type is ObjectId, which is “small, likely unique, fast to generate, and ordered.”

```
{ "_id" :      "37010"
  "city" :     "ADAMS",
  "pop" :      2660,
  "state" :    "TN",}
```

Documents: Structure References



Documents: Structure Embedded

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
            },
  access: {
             level: 5,
             group: "dev"
           }
}
```

Embedded sub-document

Embedded sub-document

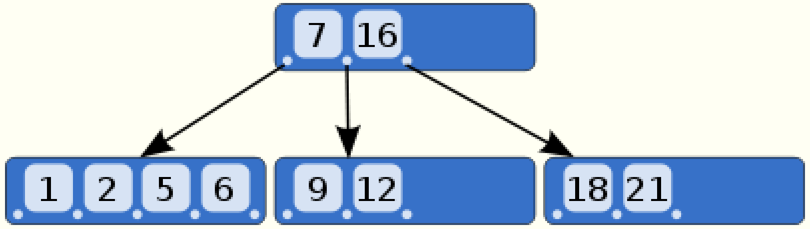
RDB Concepts to Document DB

RDBMS		MongoDB
Database	⇨	Database
Table, View	⇨	Collection
Row	⇨	Document (BSON)
Column	⇨	Field
Index	⇨	Index
Join	⇨	Embedded Document
Foreign Key	⇨	Reference
Partition	⇨	Shard

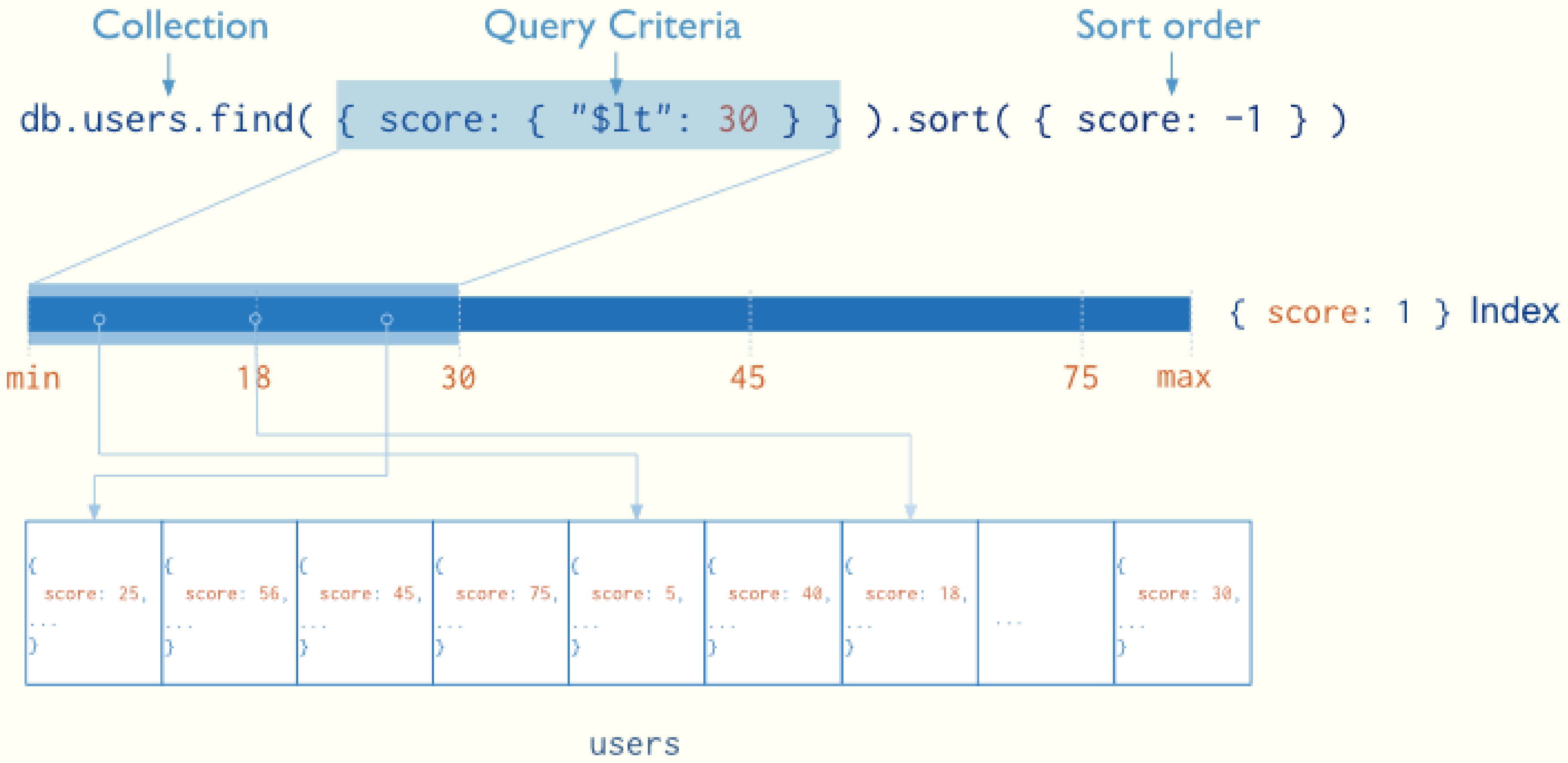
Documents: Indexing

- Indexes allows efficient queries on MongoDB.
- They are used to limit the number of documents to inspect (Otherwise, scan every document in a collection)
- By default MongoDB create indexes only on the _id field

- Indexes are created using B-tree and stores data of fields ordered by values.
- In addition, returns sorted results by using the index.



Documents: Indexing



CRUD

- Create
  - db.collection.insert(<document>)
  - db.collection.save(<document>)
  - db.collection.update(<query>, <update>, { upsert: true })
- Read
  - db.collection.find(<query>, <projection>)
  - db.collection.findOne(<query>, <projection>)
- Update
  - db.collection.update(<query>, <update>, <options>)
- Delete
  - db.collection.remove(<query>, <justOne>)

CRUD Example

```
> db.user.insert({
     first: "John",
     last : "Doe",
     age: 39
})
```

```
> db.user.find ()
{
     "_id" : ObjectId("51…"),
     "first" : "John",
     "last" : "Doe",
     "age" : 39
}
```

```
> db.user.update(
     {"_id" :
ObjectId("51…")},
     { $set: {
            age: 40,
            salary: 7000}
     }
)
```

```
> db.user.remove({
     "first": /^J/
})
```

Query Interface

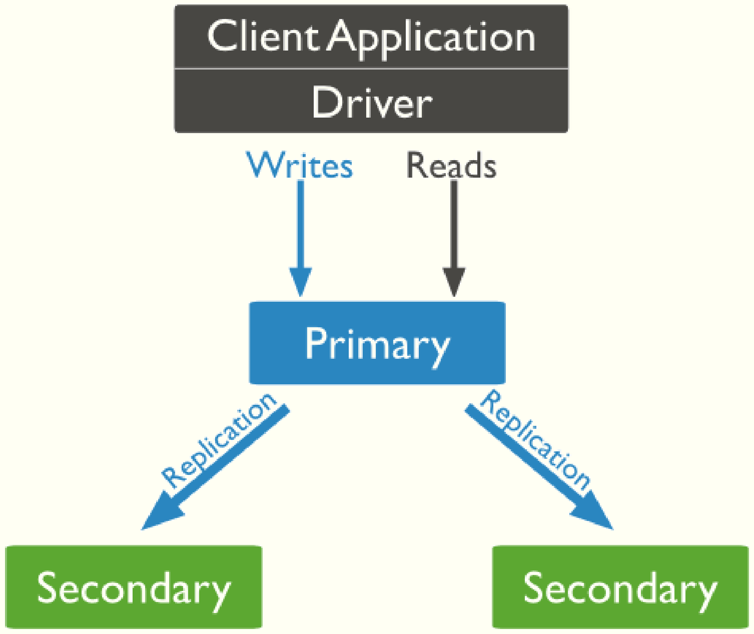
`db.users.find(`	collection
`{ age: { $gt: 18 } },`	query criteria
`{ name: 1, address: 1 }`	projection
`).limit(5)`	cursor modifier

`SELECT _id, name, address`	projection
`FROM users`	table
`WHERE age > 18`	select criteria
`LIMIT 5`	cursor modifier

Replication of Data

- Ensures redundancy, backup, and automatic failover
  - Recovery manager in the RDMS
- Replication through groups of servers known as replica sets
  - Primary set – set of servers that client tasks direct updates to
  - Secondary set – set of servers used for duplication of data

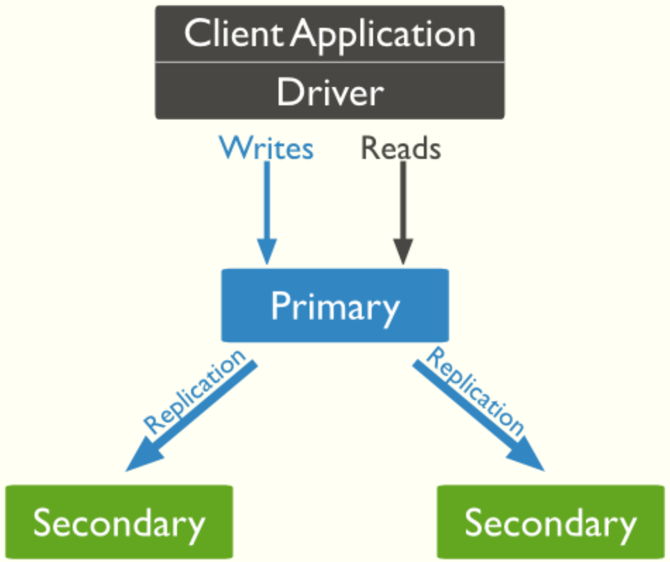
  - If the primary set fails the secondary sets 'vote' to elect the new primary set



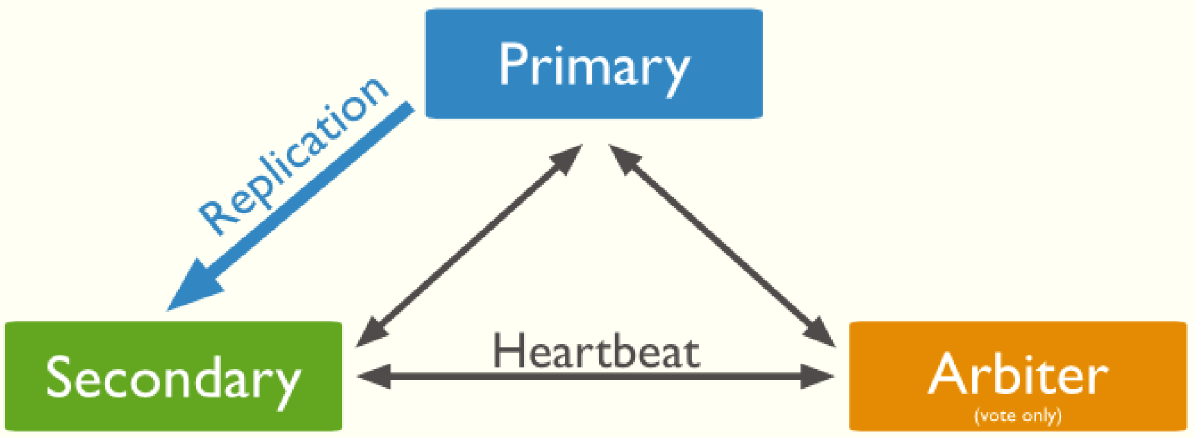
Scaling: Heavy Reads

- Scaling is achieved by adding more read slaves
- All the reads can be directed to the slaves.
- When a node is added it will sync with the other nodes -- no need to stop the cluster.

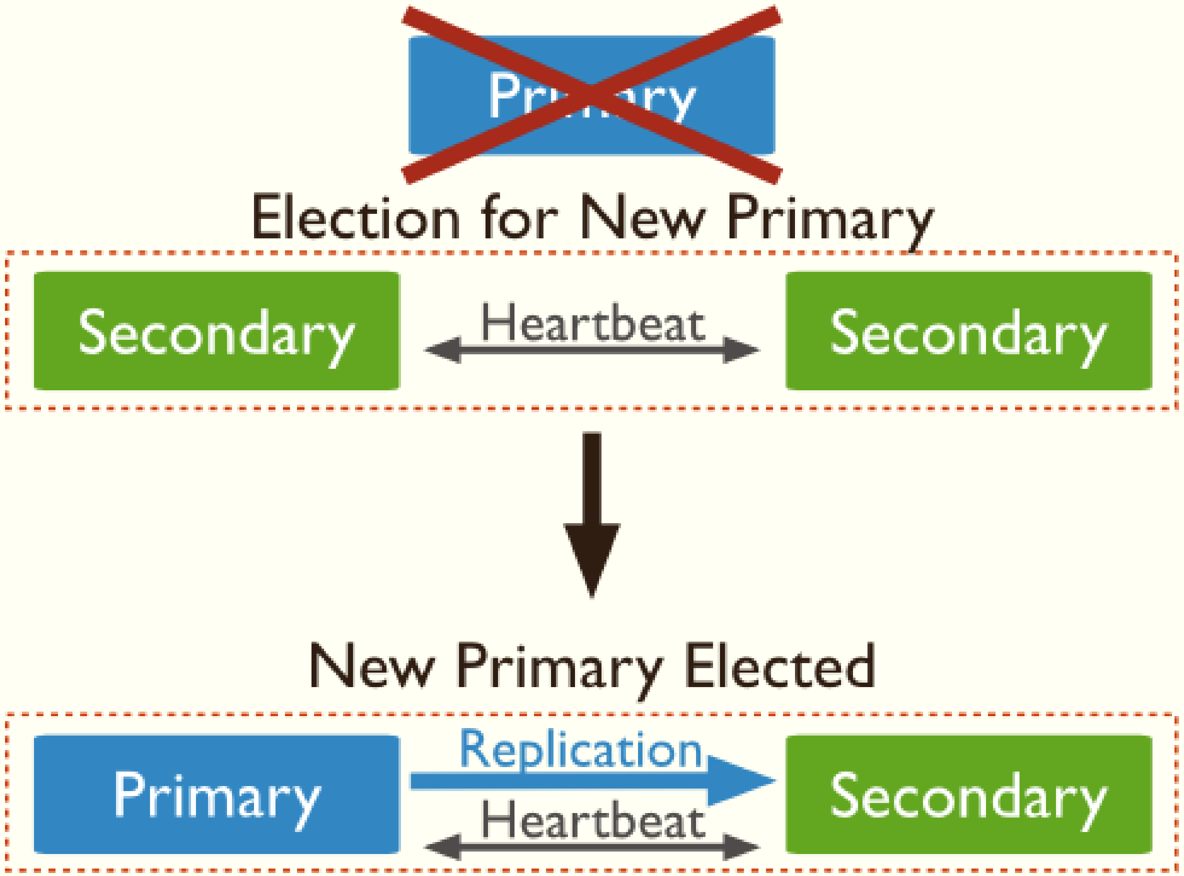
rs.add(“mongo_address:27017”)



Data Replication

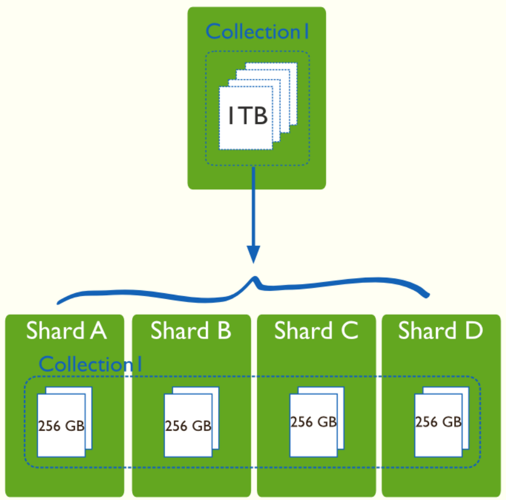


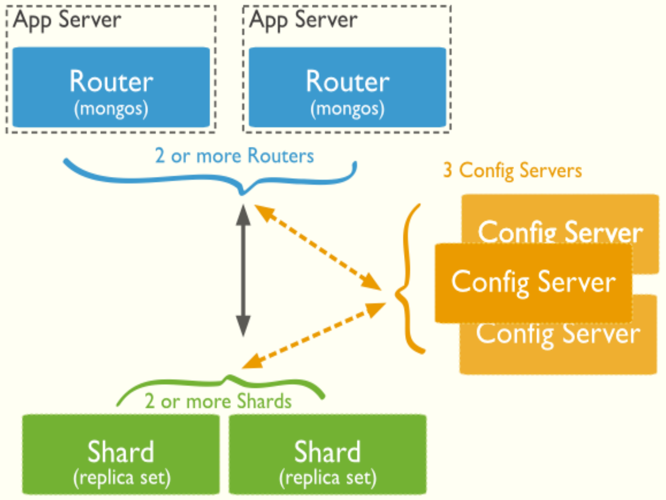
Automatic Failover



Sharding in MongoDB

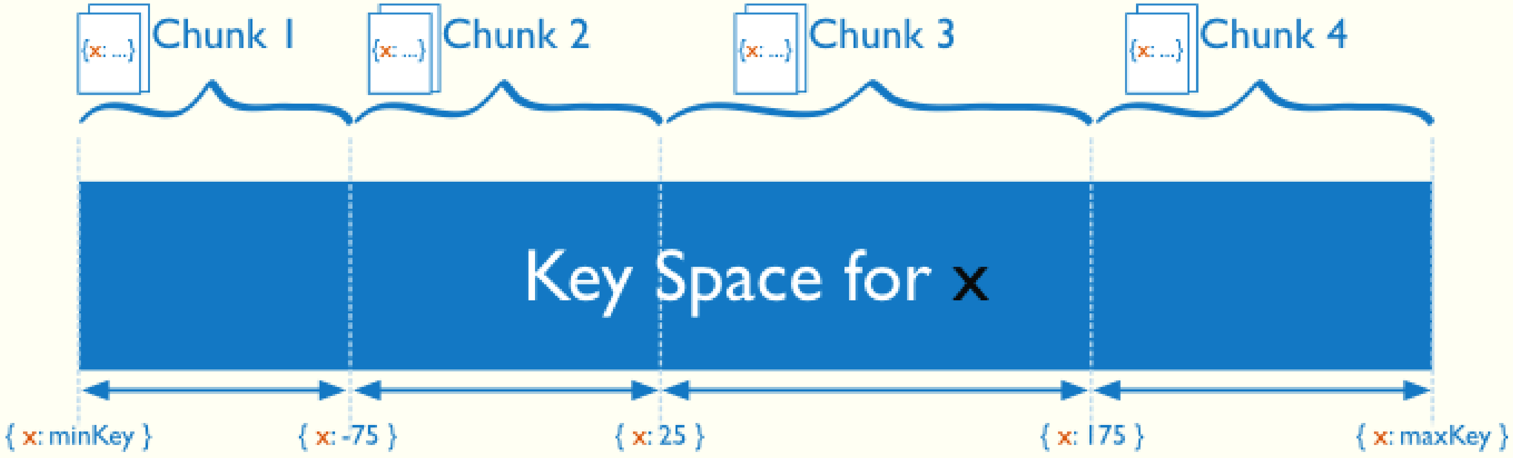
- Sharding, or horizontal scaling divides the data set and distributes the data over multiple servers.
- Each shard is an independent database, and collectively, the shards make up a single logical database.
- Query Routers: interface to client and direct queries
- Config Server: store cluster's metadata.





Range Based Sharding

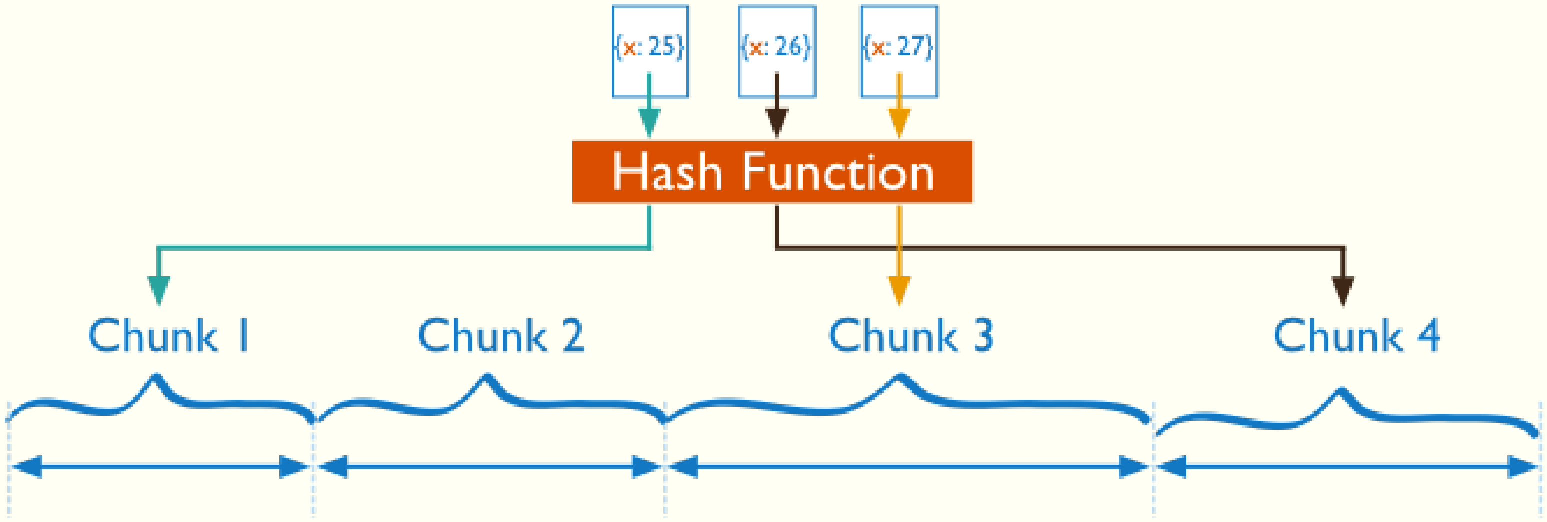
- divides the data set into ranges determined by the shard key values to provide range based partitioning.



- supports more efficient range queries
- However, result in an uneven distribution of data.

Hash Based Sharding

- computes a hash of a field's value, and then uses these hashes to create chunks



- More likely to ensure even distribution of data at the expense of efficient range queries.

Document Store: Advantages

- Documents are independent units
- Application logic is easier to write. (JSON).
- Schema Free:
  - Unstructured data can be stored easily, since a document contains whatever keys and values the application logic requires.
  - In addition, costly migrations are avoided since the database does not need to know its information schema in advance.

Suitable Use Cases

- **Event Logging:** where we need to store different types of event (order_processed, customer_logged).
- **Content Management System:** the schema-free approach is well suited
- **Web analytics or Real-Time Analytics:** useful to update counters, page views and metrics in general.

When Not to Use

- Complex Transactions:
  - atomic cross-document operations

- Queries against Varying Aggregate Structure:
  - i.e., when the structure of the aggregates vary because of continuous data evolutions

Graph Database

Motivations

- The necessity to represent, store and manipulate complex data make RDBMS somewhat obsolete
- Problem 1: Violations of the 1NF
  - Multi-valued attributes
  - Complex attributes
- Problem 2 : Accommodate Changes
  - acquiring data from autonomous dynamic sources or Web
  - RDBMS require schema renormalization
- Problem 3: Unified representation for:
  - Data
  - Knowledge (Schemas are a subset of this)
  - Queries [results + def]
  - Models (Concepts)

Existing Approaches

- RDBMS
  - Need schema renormalization

- Approaches that try to fix the above-mentioned problems:
  - OO Databases [P1], [P2] - graphs [but procedural]
  - XML Databases [P1] (somewhat [P3]) – trees
  - OORDBMS [P1] – graphs with foreign keys
  - RDF triple stores [P1, P2], somewhat [P3]

- Others
  - Datalog – more efficient fragment of Prolog
  - Network Models - graphs
  - Hierarchical Models – trees

What is a Graph Database?

- A database with an explicit graph structure: Each node knows its adjacent nodes
- As the number of nodes increases, the cost of a local step (or hop) remains the same; Plus an Index for lookups
- Express Queries as Traversals. Fast deep traversal instead of slow SQL queries that span many table joins.
- Very natural to express graph related problem with traversals (recommendation engine, find shortest path etc..)
- Seamless integration with various existing programming languages.
  - Two design principle: Declarativity & Change

- Distinguish between "Database for graph as object"!

Database Representation

- Sailors(sid:integer, sname:char(10), rating: integer, age:real)
- Boats(bid:integer, bname:char(10), color:char(10))
- Reserve(sid:integer, bid:integer, day:date)

Sailors

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

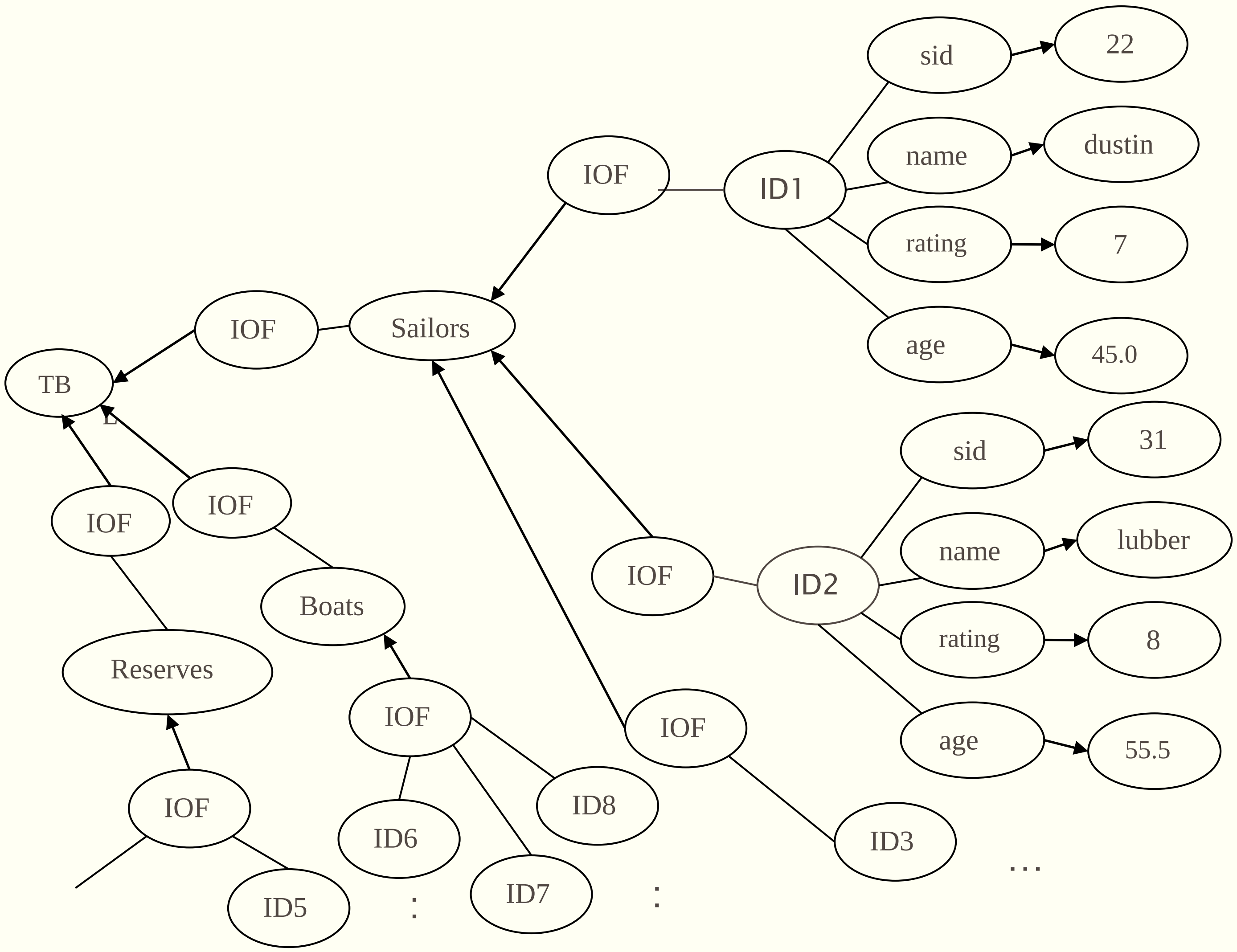
Reserves

sid	bid	day
22	101	10/10/96
58	103	11/12/96

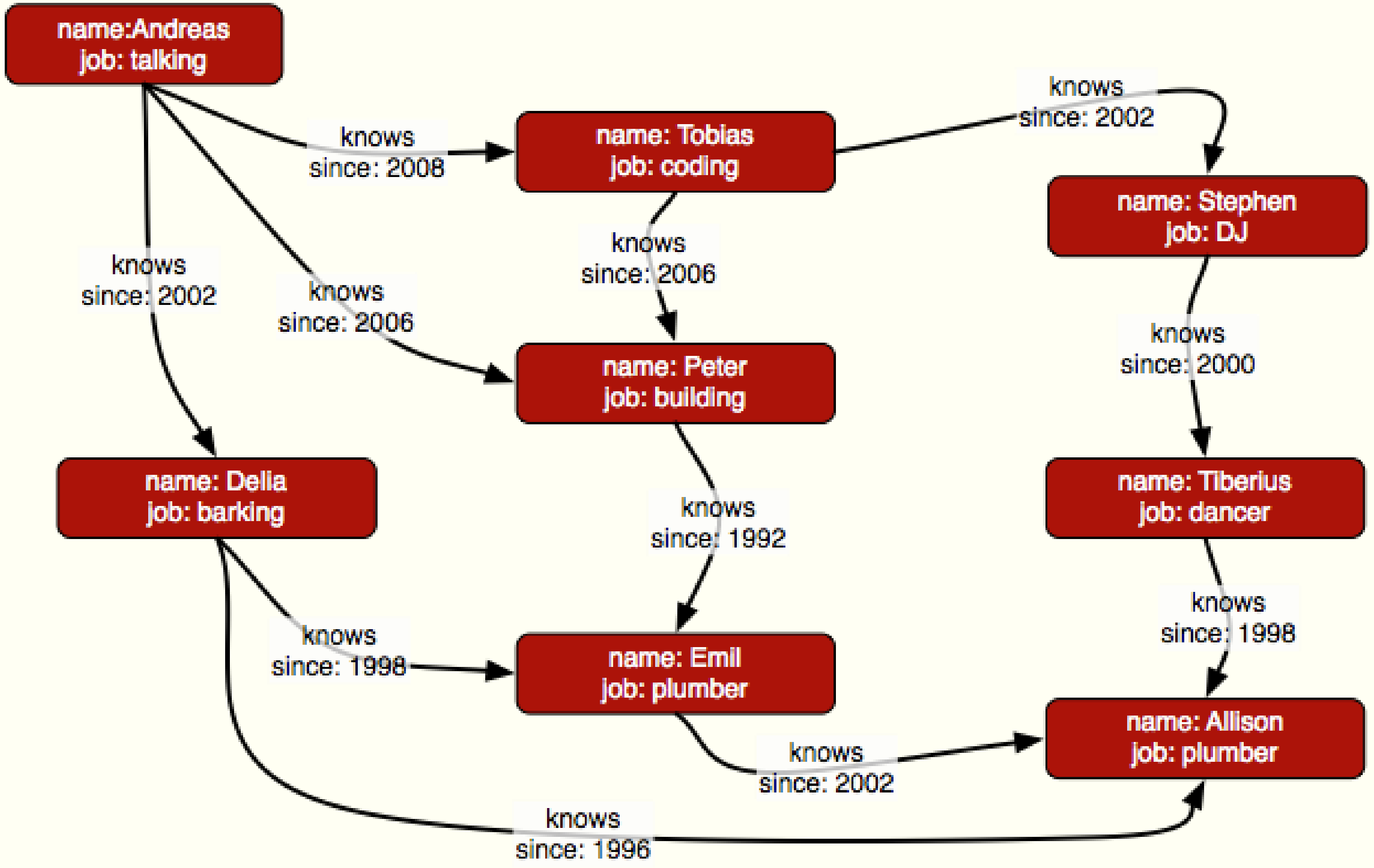
Boats

bid	bname	color
101	Interlake	red
102	Clipper	green
103	Marine	red

Graph Representation

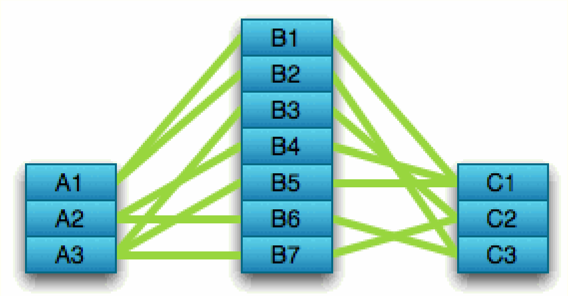


Property Graph

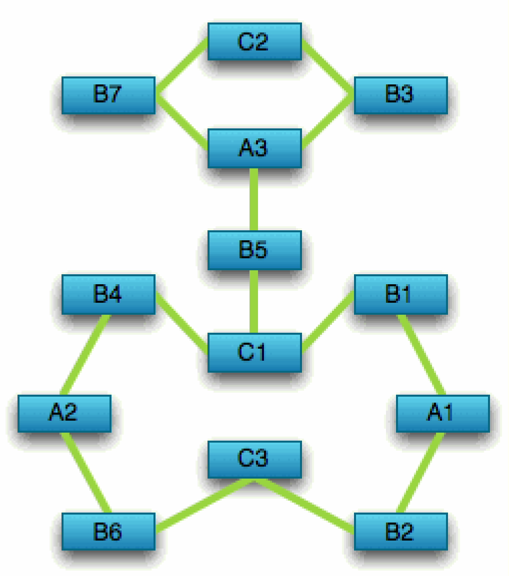


Compared to Relational Databases

Optimized for aggregation

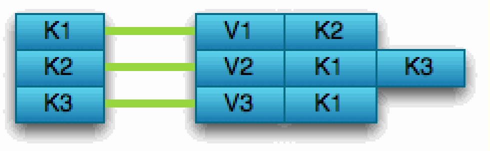


Optimized for connections

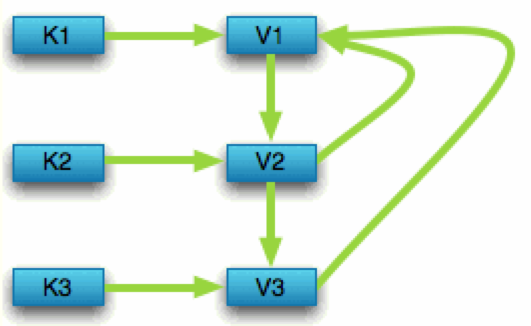


Compared to Key-Value Store

Optimized for
simple look-ups



Optimized for
traversing connected data



Social Network “path exists” Performance

- Experiment:
  - ~1k persons
  - Average 50 friends per person
  - `pathExists(a,b)` limited to depth 4

	# persons	query time
Relational database	1000	2000ms
Neo4j	1000	2ms
Neo4j	1000000	2ms

Neo4j?

- A Graph Database + Lucene Index
- Property Graph
- Full ACID (atomicity, consistency, isolation, durability) (?)
- High Availability (with Enterprise Edition)
- 32 Billion Nodes, 32 Billion Relationships, 64 Billion Properties
- Embedded Server
- REST API

Good For

- Highly connected data (social networks)
- Recommendations (e-commerce)
- Path Finding (how do I know you?)
- A* (Least Cost path)
- Data First Schema (bottom-up, but you still need to design)

Summary

- SQL Databases
  - Predefined Schema
  - Standard definition and interface language
  - Tight consistency (ACID)
  - Well defined semantics

- NoSQL Database
  - No predefined Schema
  - Per-product definition and interface language
  - Getting an answer quickly is more important than getting a correct answer (BASE)

Summary: noSQL Common Advantages

- Cheap, easy to implement (open source)
- Data are replicated to multiple nodes (therefore identical and fault-tolerant) and can be partitioned
  - Down nodes easily replaced
  - No single point of failure
- Easy to distribute
- Don't require a schema
- Can scale up and down
- Relax the data consistency requirement (CAP)

Summary: What are we giving up?

- joins
- group by
- order by
- ACID transactions (none are strict ACID!)
- SQL as a sometimes frustrating but still powerful query language
- easy integration with other applications that support SQL